Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Aplicación de machine learning en un aeropuerto: reconocimiento facial y gestión de colas

TITULACIÓN: Doble titulación de Grado en Ingeniería de Sistemas Aeroespaciales e Ingeniería Telemática

AUTOR: Víctor Piñón Rattia

DIRECTOR: Eduard Garcia Villegas

FECHA: 22 de junio de 2020

Título: Aplicación de machine learning en un aeropuerto: reconocimiento facial y gestión de colas

Autor: Víctor Piñón Rattia

Director: Eduard Garcia Villegas

Fecha: 22 de junio de 2020

Resumen

Los algoritmos de Machine learning han permitido desarrollar, durante los últimos años, una gran cantidad de aplicaciones, que son utilizadas por muchas personas a diario, estas proporcionan funcionalidades en campos tan diversos como la radiología o los asistentes virtuales controlados por voz. Muchas de estas aplicaciones, hubiesen sido prácticamente imposibles de desarrollar sin los avances realizados en esta materia durante las últimas décadas, pero con algoritmos de machine learning son posibles.

El principal objetivo de este trabajo es encontrar y explorar aplicaciones potenciales de machine learning en la gestión de un aeropuerto. Como prueba de concepto, se han estudiado en profundidad dos aplicaciones: el reconocimiento facial y la predicción del tiempo de espera en colas en aeropuertos.

En lo que respecta al reconocimiento facial, se desarrolla, implementa y evalúa un modelo de recocimiento facial *edge*, y también se implementa y evalúa un modelo de reconocimiento facial *cloud* utilizando la Face API de Azure. Ambas implementaciones se realizan en Raspberry Pi, con la intención de comparar el tiempo y consumo de energía que se requiere para cada uno de los dos modelos.

Los resultados de reconocimiento facial obtenidos son buenos en ambos casos, pero son superiores en el modelo *cloud*, tanto en calidad del reconocimiento, como en tiempo y consumo de energía, siendo la mejor opción utilizar el modelo *cloud* con una conexión a internet mediante cable.

Respecto a la herramienta para predecir el tiempo de espera en cola en un aeropuerto, se ha requerido un procesado de datos públicos sobre tiempos de esperas en aeropuertos. A continuación, se ha hecho una comparación entre varios algoritmos para ver cuál daba mejores resultados y, finalmente, se ha optimizado el algoritmo que ha dado mejores resultados.

Los resultados obtenidos con esta herramienta son prometedores, y permiten ampliar el modelo desarrollado en este trabajo, para futuras aplicaciones relacionadas con las colas en aeropuertos.

Title: Application of machine learning in an airport: face recognition and queue management

Author: Víctor Piñón Rattia

Director: Eduard Garcia Villegas

Date: June 22 nd 2020

Overview

In the last few years, Machine learning algorithms have enabled the development of many applications, which are used by a lot of people on a daily basis. This applications provide functionalities in fields as diverse as radiology or virtual assistants with voice recognition. Many of this applications, would have been nearly impossible to develop without the progress made in the field among the last decades, but are now possible with machine learning algorithms. It is also likely, because of the current growth in this topic, that we will continue to see more and more applications being developed using machine learning in the next years.

The main goal of this work is to find and explore potential applications of machine learning on the management of an airport. As a proof of concept, two particular applications have been deeply studied: face recognition and prediction of waiting times in airport queues.

Regarding face recognition, there will be an edge computing-based development, implementation and evaluation and there will also be an implementation and evaluation of a cloud-based model, which uses Azure Face API. Both implementations are done using a Raspberry Pi, with the intent of comparing the run time and energy consumption of the two approaches.

The obtained face recognition results are good in both edge and cloud models, but they are better in the cloud model, which is excellent in face recognition quality. Regarding run time and energy consumption, the cloud model also wins, with the best performance being when used with a cable internet connection between the camera-equipped end device and the network.

With respect to the tool for predicting airport passport control waiting times, the chosen methodology involved different steps. A first phase, pre-processing public data of airport waiting times. Then, a second phase in which several machine learning algorithms are tested to evaluate which one performed the best. Finally, the chosen machine learning algorithm is optimized.

The results obtained with this tool are really promising and they could be a starting point for future applications using this model in airport queues.

ÍNDICE

INTRODUCCIÓN

Actualmente, los aeropuertos tienen una gran complejidad, teniendo que procesar una gran cantidad de pasajeros, equipajes y mercancías; así como gestionar los movimientos de aeronaves y proporcionarles la asistencia y los suministros necesarios. Esta complejidad, se ha ido incrementado y probablemente continuará con esta tendencia, a medida que crezca la aviación comercial.

Debido a esta complejidad, puede resultar especialmente interesante automatizar algunas de las tareas que se deben llevar a cabo. Utilizando la información que se puede generar en un aeropuerto, se pueden utilizar algoritmos de machine learning, que podrían resultar muy útiles para la automatización del aeropuerto. Por ello, este trabajo se centrará en la aplicación de herramientas de machine learning en un aeropuerto.

Actualmente, ya hay herramientas de machine learning implementadas en algunos aeropuertos. Las más frecuentes suelen ser las relacionadas con el tratamiento y seguimiento de equipajes [1], aunque también se aplican en otros ámbitos como la predicción de la llegada de las aeronaves [2]. Aun así, todavía existen muchos ámbitos dentro de los aeropuertos, en los que las herramientas de machine learning se podrían implementar para optimizar y automatizar procesos. Entre estos ámbitos se encuentra la gestión de las colas, que representa uno de los mayores problemas que existen en los aeropuertos [3][4] y que afecta especialmente a los pasajeros.

Otra de las posibles aplicaciones de machine learning en aeropuertos, es el reconocimiento facial. Que se podría utilizar para funciones muy diversas, que van desde la seguridad, hasta acelerar el paso por el filtro de seguridad [5].

Teniendo en cuenta este contexto, los objetivos de este trabajo serán: aumentar el conocimiento sobre machine learning (tanto en técnicas, como en evaluación), estudiar las posibles aplicaciones de machine en la gestión de un aeropuerto y desarrollar algunas de estas aplicaciones. Las aplicaciones que se desarrollaran serán: reconocimiento facial, intentando encontrar la opción más eficiente energéticamente y en tiempo de proceso; y la predicción del tiempo de espera en cola en un aeropuerto.

Para encontrar la implementación de reconocimiento facial más eficiente energéticamente, se compararán dos modelos distintos: por un lado un modelo *edge*, en el cuál se realizan los cálculos en el mismo dispositivo que adquiere los datos, en este caso el que tiene la cámara; y por otro lado un modelo *cloud*, en el que los cálculos necesarios se realizan en otro dispositivo distinto al que obtiene los datos. Para implementar este modelo *coud*, se utilizará Azure, en concreto Face API, que permite detectar, reconocer y analizar caras. La implementación se hará en ambos casos utilizando hardware de bajo coste, específicamente Raspberry Pi, que es un ordenador de dimensiones y consumos reducidos, que nos permitirá comparar los dos modelos en igualdad de condiciones.

Otros de los pasos necesarios para la implementación de un modelo de reconocimiento facial son: recopilar los datos necesarios para el entrenamiento (en el caso *edge*) y la evaluación del modelo (tanto en el caso *edge*, como en el *cloud*), escoger las librerías más adecuadas e investigar qué algoritmos se deben utilizar.

En lo que respecta al modelo de predicción del tiempo de espera en cola, también se deberán recopilar datos, así como realizar un procesado de estos para que tengan el formato adecuado para entrenar exitosamente el algoritmo de inteligencia artificial. Después de evaluar y escoger el entorno de programación adecuado, se comparan los distintos algoritmos para averiguar cuál proporciona mejores resultados para este problema en concreto y, finalmente, optimizar el algoritmo escogido para que de los mejores resultados posibles.

Como se verá a lo largo del documento, se ha cumplido con los los objetivos marcados. La precisión del modelo reconocimiento facial en modo *edge*, fue del 97% en el mejor de los casos, mientras que en el modo *cloud*, fue del 100%. Respecto al consumo energético y al tiempo de ejecución, el modelo *could* también obtuvo mejores resultados, siendo la mejor opción utilizar una conexión mediante cable entre el dispositivo que obtiene las imágenes y la red. En lo referente a la predicción del tiempo de espera en cola, el error relativo obtenido ha sido del 7,9% en minutos usando un modelo de red neuronal.

Este documento se estructura de la siguiente forma: en el primer capítulo se hará una introducción sobre machine learning, explicando qué es exactamente, cuáles son los diferentes tipos de algoritmos que hay y qué son las redes neuronales. También se verá cómo se evalúan los algoritmos de machine learning. En el segundo capítulo se explicará el funcionamiento de un modelo de reconocimiento facial *edge*, se implementará en una Raspberry Pi y se evaluará este modelo. En el tercer capítulo se explicará e implementará otro modelo, esta vez basado en el concepto de *cloud*, que también se evaluará y se comparará su precisión con el modelo anterior. En el cuarto capítulo, se compararán los dos modelos de reconocimiento facial en consumo energético y tiempo. En el quinto capítulo, se desarrollará una herramienta que permitirá predecir el tiempo de espera en la cola de control de pasaportes de un aeropuerto. Para el desarrollo de esta herramienta, se deberán encontrar y procesar los datos, seleccionar el algoritmo adecuado y optimizar este algoritmo. Finalmente, en el sexto capítulo se extraerán las conclusiones de este trabajo.

CAPÍTULO 1. MACHINE LEARNING

En este capítulo se explicará qué es machine learning, en concreto cuáles son los principales tipos y las herramientas que se utilizarán en los siguientes capítulos.

1.1. ¿Qué es machine learning?

Machine learning (aprendizaje automático) es una rama de la inteligencia artificial, que se basa en la idea de que hay algoritmos que nos permiten obtener resultados útiles e interesantes a partir de un conjunto de datos, sin tener que escribir un código específico para ese problema. En lugar de escribir código, se le proporcionan datos al algoritmo que crea su propia lógica basándose en los datos [6].

Por ejemplo, como se muestra en la **Fig 1.1**, un tipo de algoritmo puede ser de clasificación, que se podría utilizar para reconocer números escritos a mano, o también se podría utilizar para clasificar correos electrónicos, en correo deseado o correo no deseado (spam). Todo esto se puede hacer con el mismo algoritmo y sin necesidad de cambiar el código, solo siendo necesario cambiar los datos que se le proporcionan al algoritmo.

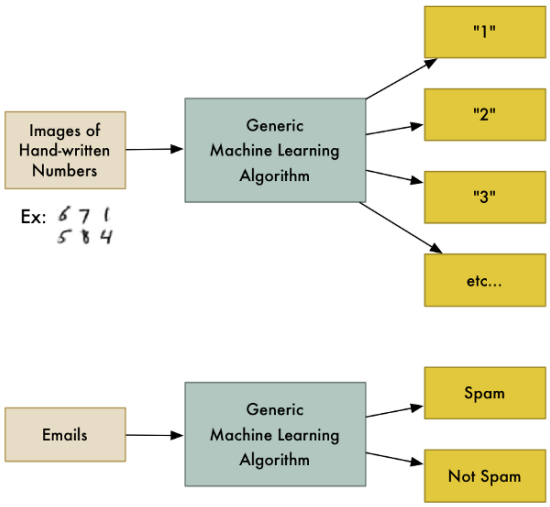

Fig. 1.1 Algoritmo de clasificación con múltiples usos [1]

1.2. Tipos de algoritmos de machine learning

Los algoritmos de machine learning se pueden dividir en tres categorías principales: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

1.2.1. Aprendizaje supervisado

En el aprendizaje supervisado, se poseen una serie de datos que relacionan valores de entrada con valores de salida, esta salida pueden ser números o etiquetas de clasificación. El objetivo de los algoritmos de aprendizaje supervisado es que, después de ser entrenados con datos de muestra, sean capaces de dar unos valores de salida en función de unos datos de entrada, tal y como lo haría el sistema real que se pretende modelar, o de la forma más parecida posible.

Algunos ejemplos de aprendizaje supervisado serían los casos descritos anteriormente, reconocer números escritos a mano, donde el valor de salida sería un valor numérico, o separar correos electrónicos en correo deseado y no deseado, donde el valor de salida sería una etiqueta de clasificación.

Dentro de los algoritmos de aprendizaje supervisado se pueden encontrar dos principales tipos:

- Clasificación: donde el resultado es una etiqueta de clasificación dependiendo de los valores de entrada. Un ejemplo sería el caso que se explicó anteriormente de separación de correos electrónicos, en correo deseado y no deseado.
- Regresión: donde el resultado es un valor numérico que intenta modelar un sistema real. Un ejemplo sería determinar el precio de una vivienda dependiendo de sus características (ubicación, habitaciones, superficie,…).

1.2.2. Aprendizaje no supervisado

En el aprendizaje no supervisado, se tienen valores de entrada pero no están relacionados con valores de salida. El objetivo es encontrar patrones en los datos que se le proporcionan al algoritmo, e incluso poder clasificar los datos en diferentes categorías, que no se conocían previamente.

Para poder clasificar los datos, se puede utilizar el concepto de *clustering* [7], que consiste en agrupar datos en grupos, según las características que tienen estos datos. Cada grupo de datos con similares características se denomina *cluster*.

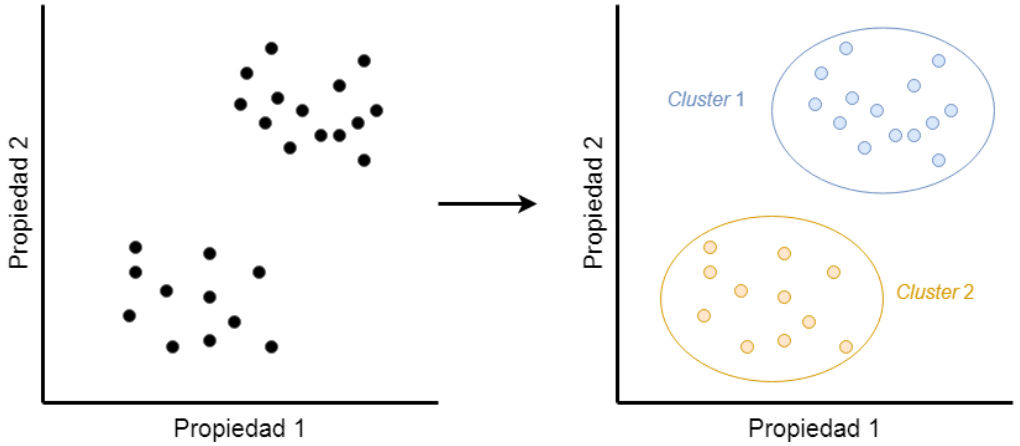

Fig. 1.2 Clustering

Un ejemplo de aprendizaje no supervisado, podría ser el que se ve en la siguiente figura **Fig. 1.2**, donde se tienen una serie de objetos mezclados como datos de entrada, que no están relacionados con ninguna etiqueta, ni ordenados de ninguna manera. Una vez estos datos salen del algoritmo, se separan en los diferentes tipos de objetos que son, pero sin etiquetarlos, ya que el algoritmo los agrupa dependiendo de los valores de entrada y las similitudes o diferencias que es capaz de encontrar.

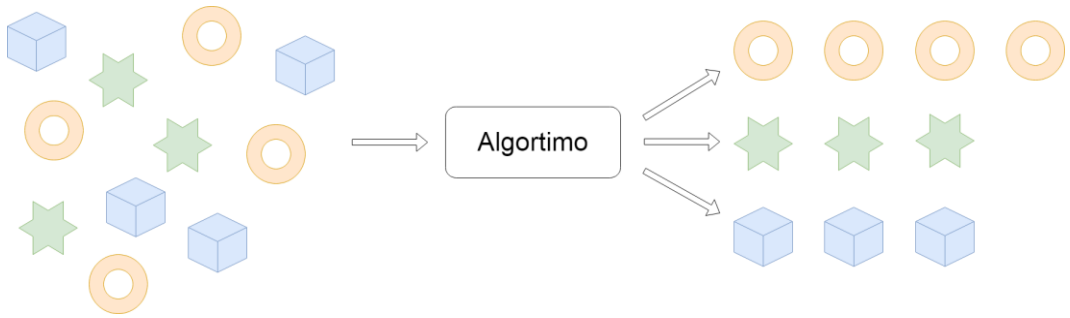

Fig. 1.3 Algoritmo de aprendizaje no supervisado

1.2.3. Aprendizaje por refuerzo

En el aprendizaje por refuerzo, como se muestra en la **Fig 1.4**, no se tienen valores de entrada ni de salida para introducir en el algoritmo, sino que hay un agente que debe realizar una acción, a la cual su entorno le devolverá una respuesta positiva o negativa. En función de las acciones que el agente realiza, el algoritmo va aprendiendo, para optimizar el resultado final de las respuestas.

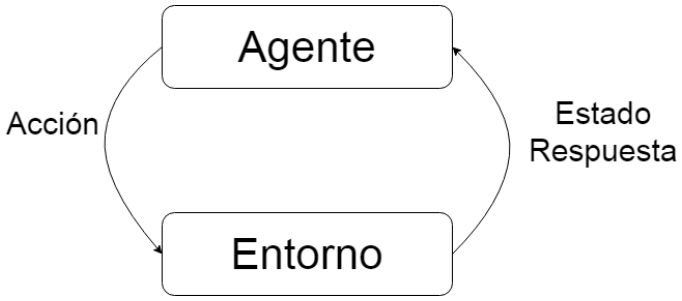

Fig. 1.4 Algoritmo de aprendizaje por refuerzo

Es importante tener en cuenta que el entorno varía en función de las acciones que toma el agente, de manera que puede haber situaciones en las que realizar una acción que tenga como resultado una respuesta negativa, cree la posibilidad de obtener mejores respuestas positivas en un futuro, cambiando el estado del entorno, y acabe mejorando el resultado.

1.3. Ejemplos de algoritmos

En este apartado se explicarán dos de los algoritmos más utilizados en machine learning. Se explicará primero la regresión lineal múltiple, a modo de ejemplo introductorio de algoritmo de machine learning y a continuación, las máquinas de vectores de soporte, que se utilizarán posteriormente en el apartado 2.1.3.

1.3.1. Regresión lineal múltiple

Un ejemplo de algoritmo que se utiliza en machine learning es la regresión lineal múltiple. Una regresión lineal es un algoritmo de aprendizaje supervisado, en el cual se introducen una serie de datos que se componen de uno (regresión lineal simple) o más (regresión lineal múltiple) valores de entrada que se relacionan con un valor de salida, que siempre es un valor numérico.

Por ejemplo, si se tiene una colección de datos que son información sobre viviendas y sobre cada vivienda se conocen las siguientes variables de entrada: número de habitaciones, superficie y barrio donde está situada, como se muestra en la **Tabla 1.1**. Estos valores van relacionados con el valor de salida que es el precio de venta de la vivienda.

Tabla 1.1 Datos de las viviendas [1]

Habitaciones	Superficie	Barrio	Precio de venta
3	185 m^2	Normal	250.000 €
2	75 m^2	Caro	300.000 €
2	80 m^2	Normal	150.000 €
1	50 m^2	Normal	78.000 €
4	185 m^2	Barato	150.000 €

Para poder hacer que el algoritmo sea capaz de obtener el valor de salida a partir de los valores de entrada, se asignan pesos (w1, w2 y w3) a los diferentes valores de la siguiente manera:

$$precio = número\ de\ habitaciones * w1 + superficie * w2 + barrio * w3 \quad \textbf{(1.1)}$$

Los pesos deben ser los mismos para todas las viviendas.

El algoritmo funciona de la siguiente manera:

1. Se asigna un valor aleatorio a todos los pesos.
2. Se calcula el error cometido al calcular el precio de cada vivienda, restando el precio de venta real del valor estimado. Cuando se ha hecho con todas las viviendas, se calcula el error total.

Tabla 1.2 Errores en el precio [6]

Habitaciones	Superficie	Barrio	Precio de venta	Suposición	Error
3	185 m^2	Normal	250.000 €	178.000 €	5,18 $x10^9$
2	75 m^2	Caro	300.000 €	371.000 €	5,04 $x10^9$
2	80 m^2	Normal	150.000 €	148.000 €	4 $x10^6$
1	50 m^2	Normal	78.000 €	101.000 €	5,29 $x10^8$
4	185 m^2	Barato	150.000 €	121.000 €	8,41 $x10^8$
				Error total	**2.31 $x10^9$**

Una forma de calcular el error total del modelo, es mediante el error cuadrático medio, en el que se eleva al cuadrado el error de cada vivienda, se suman todos los errores elevados al cuadrado de todas las viviendas y finalmente, se divide el valor obtenido entre el número total de viviendas, como se observa en la **Tabla 1.2**.

3. Se repite el paso dos con todas las combinaciones posibles de pesos.
4. Finaliza cuando se encuentran los valores de los pesos que minimizan el error total.

De esta manera se obtiene un algoritmo capaz de calcular los precios para futuras viviendas que puedan estar en venta, solo con introducir los datos de la vivienda como valores de entrada en el algoritmo.

1.3.1.1. Overfitting: problema de sobreajuste

El sobreajuste (*overfitting*) es un problema que suele ocurrir en casos como el ejemplo anterior, sucede cuando se sobreentrena el algoritmo con una colección de datos, de manera que el algoritmo sea capaz de encontrar los valores de salida con mucha exactitud, para cualquier valor de los datos con los que fue entrenado, pero que sea incapaz de dar resultados correctos si se le proporcionan valores de entrada distintos, como se muestra en la **Fig 1.5**.

También puede existir el subajuste (*underfitting*), que sería cuando el algoritmo no se entrena lo suficiente y no da datos suficientemente precisos, ni para los datos con los que fue entrenado, ni para ningunos otros.

Para prevenir este problema, es necesario verificar que el algoritmo funciona correctamente con otra colección de datos distinta a la usada para el entrenamiento, antes de empezar a utilizarlo en aplicaciones reales. El método que se utiliza comúnmente para solucionar éstos problemas se denomina validación cruzada y será explicado en más detalle en el apartado 1.6.1.

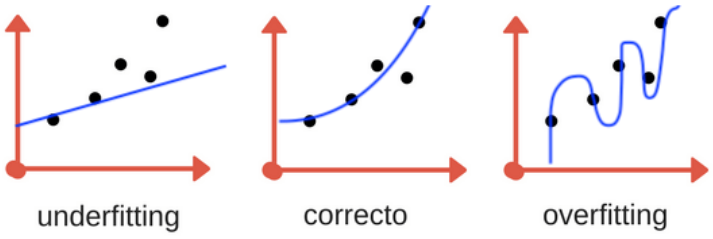

Fig. 1.5 Sobreajuste y subajuste [3]

1.3.2. Máquinas de vectores de soporte (Support vector machine, SVM)

Una máquina de vectores de soporte es un algoritmo de machine learning, de aprendizaje supervisado, que se utiliza para clasificar datos. Será utilizada

posteriormente en el apartado 2.1.3 para realizar el reconocimiento facial en este proyecto.

El funcionamiento es el siguiente: con los datos que recibe para entrenarse, el algoritmo separa los datos representándolos en función de los valores asociados a estos, de manera que le permita tenerlos separados en *clusters* (grupos de datos) diferentes. Una vez hecho esto, se trazan líneas paralelas en los puntos más cercanos entre diversos *clusters*, que hacen de frontera entre ellos. También se traza una línea media entre las paralelas.

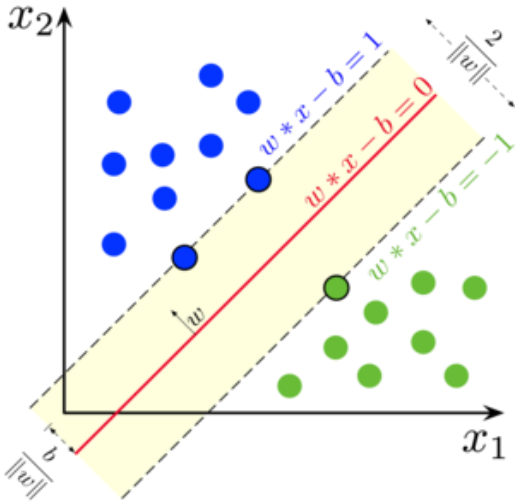

Fig. 1.6 Funcionamiento SVM [4]

La figura **Fig. 1.6** representa un caso en el que se tienen datos definidos por dos parámetros: *x1* y *x2*. Se puede observar como estos datos se pueden separar en dos *clusters* que serían el *cluster* azul y el *cluster* verde. También están representadas las líneas paralelas mencionadas anteriormente que tocan los puntos más cercanos entre los dos *clusters*, así como la línea media entre estas paralelas. Este espacio entre las líneas paralelas hace de frontera entre los dos *clusters* para poder clasificar los datos recibidos posteriormente. El vector marcado como *ω* es conocido como vector de soporte y es el que da nombre a este modelo.

Una vez el algoritmo ha sido entrenado y ha encontrado los vectores de soporte más distanciados entre sí, es capaz de clasificar nuevos datos viendo a qué *cluster* pertenecen, dependiendo de a qué lado estén situados de la línea media, o lo que es lo mismo, a qué *cluster* se aproximan más.

1.3.2.1. ¿Qué sucede cuando no se pueden separar los datos con rectas?

En las ocasiones en las que los datos no se pueden separar con líneas rectas, se utilizan los llamados *kernels*.

Los *kernels* son operaciones que transforman los datos que se introducen para entrenar el algoritmo, de manera que añaden una dimensión y facilitan la separación en *clusters*. Algunos ejemplos de los *kernels* más habituales serian:

- Lineales
- Polinómicos
- Gaussianos
- Tangente hiperbólica

Un ejemplo de uso de un *kernel* puede ser el siguiente:

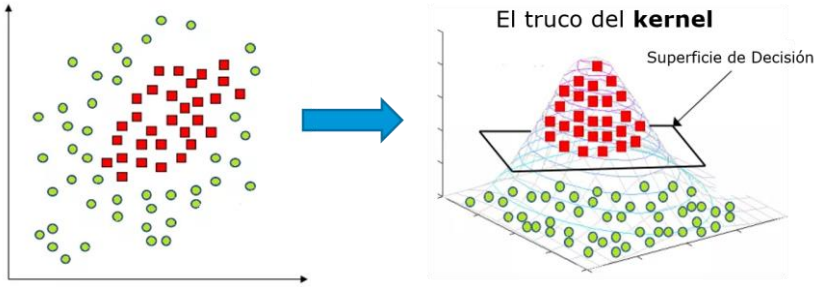

Fig. 1.7 Uso de un *kernel* [5]

Se puede ver en la **Fig 1.7**, como se pasa de una representación en dos dimensiones que no se puede separar con líneas rectas, a una imagen en tres dimensiones que sí se puede separar, pero con una superficie de decisión en lugar de con una línea recta.

1.4. Redes neuronales

En este apartado se explicarán las redes neuronales, en concreto qué es una neurona (dentro de una red neuronal) y qué es una red neuronal.

1.4.1. ¿Qué es una neurona en una red neuronal?

Utilizando el ejemplo anterior de regresión lineal múltiple, se podría cambiar ligeramente el algoritmo para simplificar los cálculos y obtener mejores resultados.

Si en lugar de calcular todas las posibles combinaciones de pesos que minimizan el error total, se calculan los pesos que minimizan el error de cada vivienda, se obtendría algo como lo que se muestra en la **Fig 1.8** para cada vivienda.

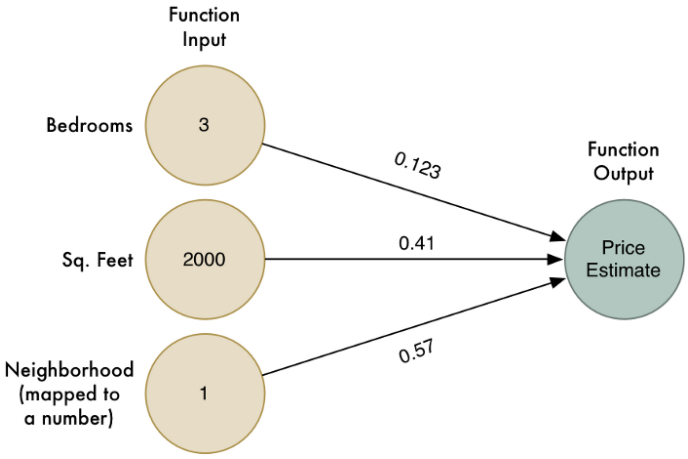

Fig. 1.8 Pesos para los valores de entrada [6]

Se debe repetir este proceso para todas las viviendas, obteniendo así un algoritmo que calcula el precio estimado para cada una de ellas, a cada uno de estos algoritmos se le llama neurona, como se muestra en la **Fig 1.9**.

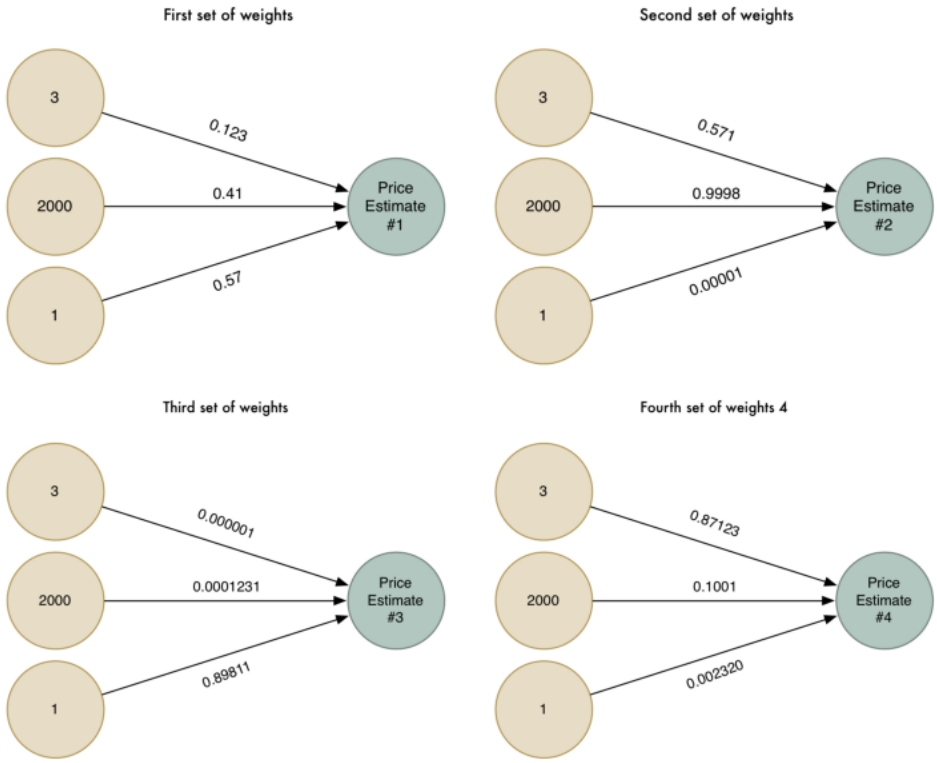

Fig. 1.9 Pesos para cuatro viviendas [6]

Finalmente se pueden combinar estas neuronas en un resultado final que proporcione el precio final de una vivienda, pero teniendo en cuenta los valores

de salida de las neuronas que ya han sido obtenidos, como se muestra en la **Fig 1.10**.

Fig. 1.10 Obtener un precio final [6]

Con este sistema se puede obtener el mismo resultado que en el ejemplo del apartado 1.3.1 pero con menor coste computacional y mayor precisión, ya que los casos más alejados de la media obtendrán una mayor importancia en los cálculos y por tanto mayor probabilidad de obtener resultados correctos.

1.4.2. ¿Qué es una red neuronal?

Una red neuronal consiste en combinar y conectar diversas neuronas en una red, como se muestra en la **Fig 1.11**.

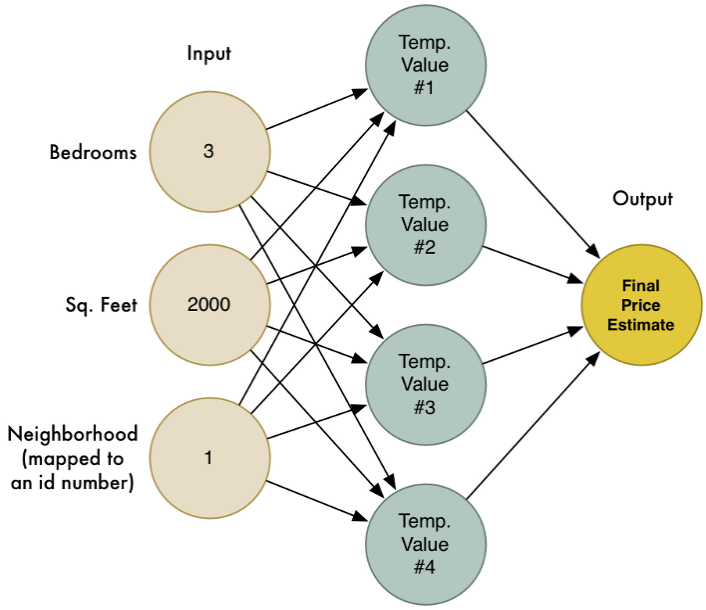

Fig. 1.11 Red neuronal [6]

Cada neurona (nodo) de la red obtiene valores de entrada, a los cuales aplica su algoritmo y envía valores de salida al siguiente nodo. De esta manera, en el ejemplo anterior, se obtiene finalmente un precio estimado para cada vivienda a la salida de la red neuronal.

1.5. Redes neuronales con múltiples capas de nodos

Una de las formas más utilizadas de crear redes neuronales, consiste en unir diferentes capas de algoritmos, de manera que cada capa superior recibe información de las capas inferiores, con los que puede dar resultados más específicos que la anterior. Esto permite obtener resultados para problemas más complejos que el mostrado en la **Fig. 1.8** donde solo había una capa de algoritmos.

En la **Fig 1.12** se puede ver un ejemplo de una red neuronal de múltiples capas de nodos:

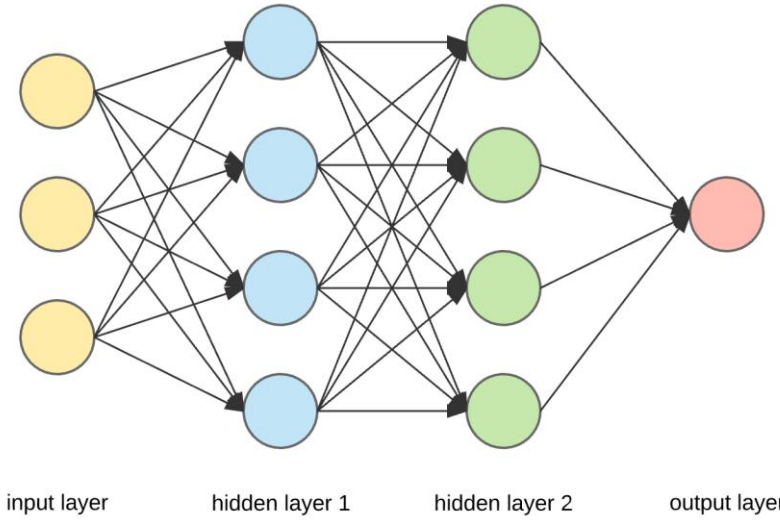

Fig. 1.12 Red neuronal con múltiples capas de nodos [7]

Una gran diferencia entre una red neuronal de múltiples capas y el ejemplo del apartado 1.4.1, es que en una red de múltiples capas de nodos el usuario no conoce los valores de entrada, ni de salida de cada nodo de la red, ya que el objetivo es maximizar la precisión de los resultados finales. Por lo tanto, el cómo se entrene específicamente cada nodo de la red, no es de interés para el usuario.

Este tipo de redes neuronales son conocidas comercialmente como redes de *Deep Learning*.

1.6. ¿Cómo evaluar un algoritmo de machine learning?

En este apartado se explicará cómo se debe realizar una correcta evaluación de un algoritmo de machine learning, para poder compararlo con otros y determinar si es lo suficientemente apropiado para la aplicación que se desee.

Para evaluar un algoritmo de machine learning se deben dividir los datos disponibles en dos grupos principales [8]:

- Grupo de entrenamiento (*Training set*): es el grupo de datos que será utilizado para entrenar el modelo.
- Grupo de validación (*Validation set*): es el grupo de datos que será utilizado para realizar las pruebas necesarias, para comprobar que el modelo sea adecuado para la aplicación que se le quiera dar.

1.6.1. Validación cruzada (cross-validation)

En algunos casos en los que no se dispone de muchos datos, es recomendable aprovechar lo máximo posible los datos de los que se dispone, un ejemplo de esto es la validación cruzada.

En la validación cruzada (*cross-validation*) se utilizan todos los datos de los que se dispone en múltiples ocasiones, tanto para entrenar como para validar. Para ello se siguen los siguientes pasos:

1. Se dividen los datos de los que se disponen en *n* grupos iguales, con datos aleatorios de los que se dispone.
2. Se entrena el algoritmo con *n-1* grupos de datos. Dejando así un grupo para el paso siguiente.
3. Una vez el algoritmo ha sido entrenado, se valida con el único grupo con el que no ha sido entrenado, como se muestra en la **Fig 1.13**.

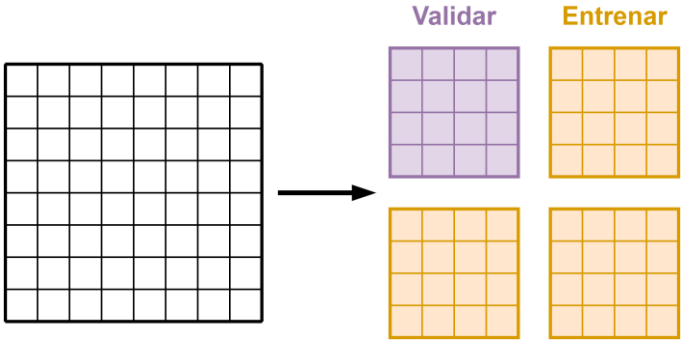

Fig. 1.13 Datos para entrenar y validar

4. Se repiten los pasos 2 y 3 *n* veces, de manera que todos los grupos de datos hayan sido utilizados para validar el algoritmo al menos una vez y hayan sido utilizados para entrenar *n-1* veces, como se muestra en la **Fig 1.14**.

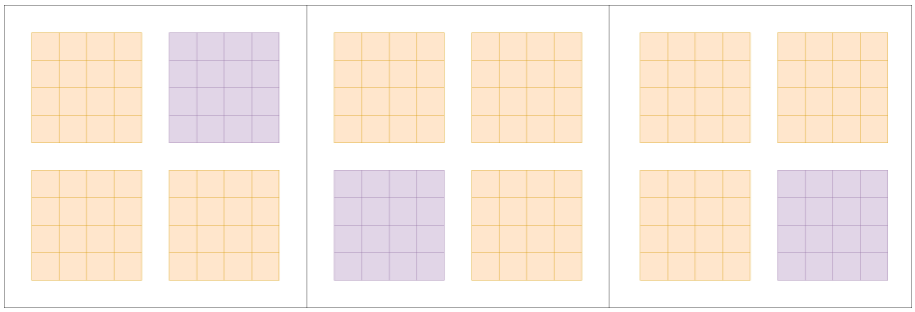

Fig. 1.14 Repetición del proceso para n=4

5. Los resultados finales se obtienen de promediar los resultados de todas las validaciones realizadas.

De esta manera, se consigue sacar la mayor cantidad de información de los datos de los que se dispone para poder entrenar y validar el modelo, siendo especialmente útil para casos en los que se tengan pocos datos.

1.6.1.1. Ejemplo con fotografías

Suponiendo que se tienen una serie de fotografías de una persona que se quiere identificar y otra serie de fotografías con caras de personas desconocidas, como se observa en la **Fig 1.15**.

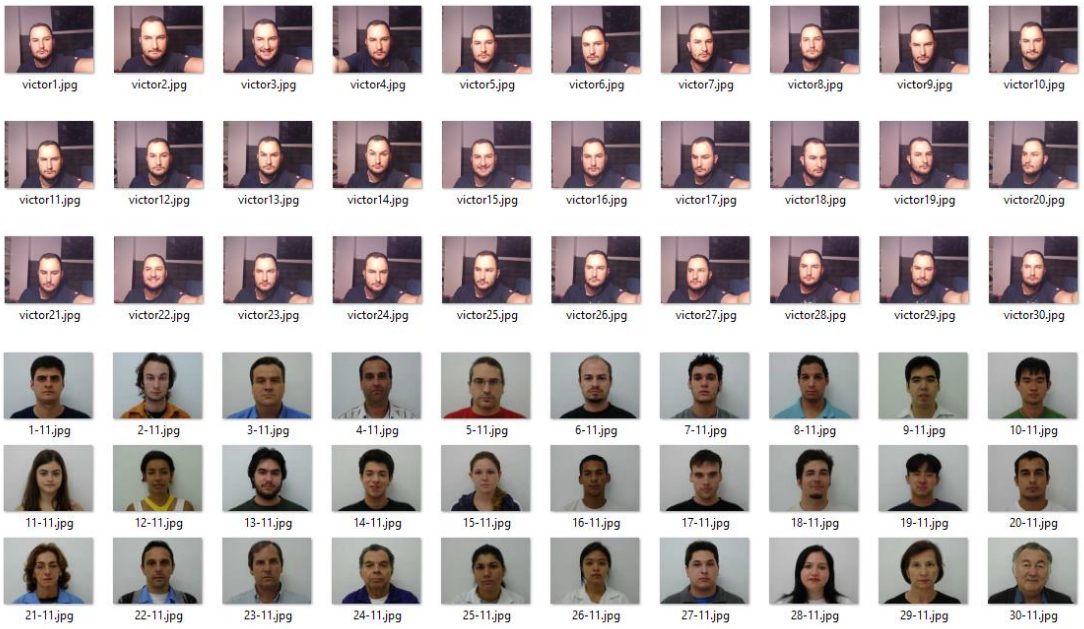

Fig. 1.15 Fotografías disponibles

1. Se dividen los datos, aleatoriamente en un número determinado de grupos, por ejemplo tres. En este caso se separarían en seis grupos, tres con fotografías de la persona que se quiere identificar y tres con fotografías de otras personas, como se muestra en la **Fig 1.16**.

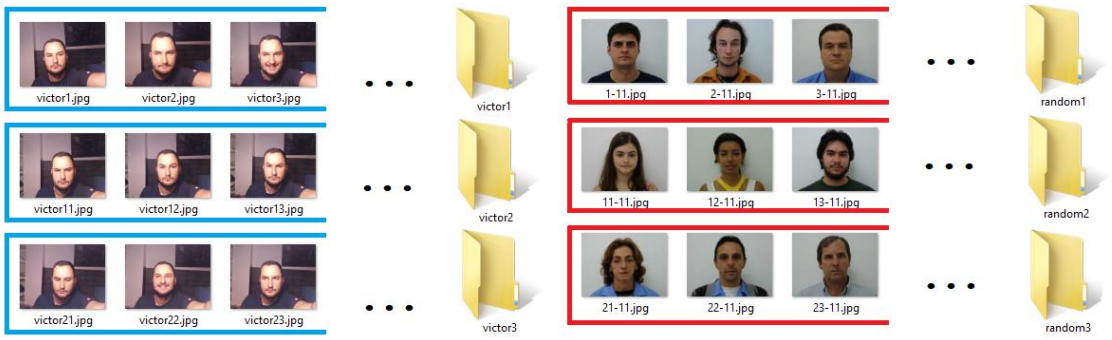

Fig. 1.16 Organización de las fotografías

2. A continuación, se entrena el algoritmo con dos de los tres grupos que pertenecen a la persona que se quiere identificar y dos de las tres carpetas de otras personas.
3. Una vez entrenado el algoritmo, se prueba (valida) con los grupos que han quedado, es decir, una para la persona que se quiere identificar y otra con las otras personas, como se muestra en la **Fig 1.17**.

Fig. 1.17 Separación para la validación cruzada

4. Se repite el proceso dos veces más, como se muestra en la **Fig 1.18**, cambiando la carpeta que se había utilizado para las pruebas, por las que se utilizaron para entrenar. De esta manera todas las fotografías se habrán utilizado para hacer pruebas una vez y dos veces para entrenar.

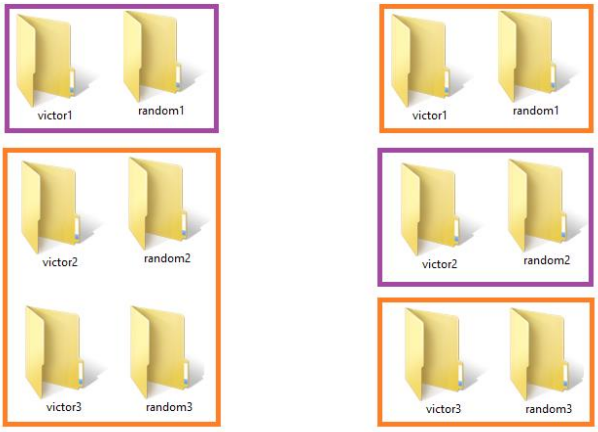

Fig. 1.18 Separación para la validación cruzada en las siguientes pruebas

5. Los resultados finales se obtienen de promediar los resultados de todos los tests realizados, en este ejemplo sería el promedio de los tres resultados obtenidos.

Con este proceso se habría realizado una validación cruzada correctamente.

1.6.2. Matriz de confusión (*confusion matrix*)

Una matriz de confusión, es la forma correcta de representar los resultados obtenidos de una validación cruzada, en la evaluación de un clasificador.

Para hacer una matriz de confusión, se separan los resultados de cada prueba de la validación cruzada en las siguientes categorías.

- TP (verdadero positivo): cuando se realizó la prueba con una imagen de la persona que se quiere identificar y el algoritmo la identificó correctamente.
- FN (falso negativo): cuando se realizó la prueba con una imagen de la persona que se quiere identificar, pero el algoritmo no la identificó correctamente, dando como resultado que no es la persona a identificar.
- TN (verdadero negativo): cuando se realizó la prueba con una imagen que no era de la persona a identificar y el algoritmo detecta que no es la persona a identificar.
- FP (falso positivo): cuando se realizó la prueba con una imagen que no era de la persona a identificar y el algoritmo la identifica como la persona a identificar, erróneamente.

Estos resultados se representan en una tabla, llamada matriz de confusión, como se muestra en la **Tabla 1.3**.

Tabla 1.3 Matriz de confusión

		Predicción	
		Conocido	Desconocido
Entrada	Conocido	TP	FN
	Desconocido	FP	TN

Una vez obtenida esta matriz de confusión se pueden calcular los siguientes valores:

- Exactitud (*accuracy*): proporción de respuestas correctas.

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN} \quad \textbf{(1.2)}$$

- Precisión (*precisión*): cuántas de las predicciones positivas son correctas.

$$\rho = \frac{TP}{TP + FP} \quad \textbf{(1.3)}$$

- Tasa de verdaderos positivos (*recall*)**:** cuántos casos positivos reales fueron identificados.

$$r = \frac{TP}{TP + FN} \quad \textbf{(1.4)}$$

Con estos valores y la matriz de confusión, se obtiene la información necesaria para poder evaluar y comparar un algoritmo supervisado de clasificación.

CAPÍTULO 2. RECONOCIMIENTO FACIAL BASADO EN OPENCV

En este capítulo se describirá el funcionamiento de un modelo de reconocimiento facial, explicando cada uno de los cuatro principales pasos de los que se compone y la relación entre ellos. Finalmente se evaluará este modelo.

2.1. Funcionamiento del modelo de reconocimiento facial

A continuación, se explicarán los diferentes pasos de los que se compone el modelo de reconocimiento facial que se utilizará en este proyecto.

2.1.1. Detección facial

El primer paso para el reconocimiento facial, es la detección facial, es decir, tener un algoritmo capaz de detectar caras dentro de una imagen.

Los algoritmos utilizados más frecuentemente para la detección facial, son algoritmos de detección de objetos, capaces de detectar cualquier objeto en una imagen, desde vehículos hasta animales, dependiendo de los datos que se utilicen para entrenar el algoritmo. Son incluso capaces de detectar múltiples objetos dentro de una imagen, sin necesidad de que estos sean del mismo tipo, como se muestra en la **Fig 2.1**.

Fig. 2.1 Detección de objetos [9]

Dentro de los algoritmos de detección de objetos, existen tres principales [9]:

- R-CNN (red neuronal convolucional recursiva) [10]: es el algoritmo más utilizado, ya que es el más preciso, pero es difícil de implementar y entrenar. También es el más lento.
- YOLO (*you only look once*) [11]: el algoritmo más rápido, aunque es el menos preciso.
- SSDs (*single shot detectors*) [12]: es el punto intermedio entre R-CNN y YOLO, proporcionando un balance entre precisión y velocidad. Es el que se utilizará en los siguientes capítulos.

En este proyecto se utilizará un *single shot detector*, que se basa en una red neuronal con múltiples capas de nodos.

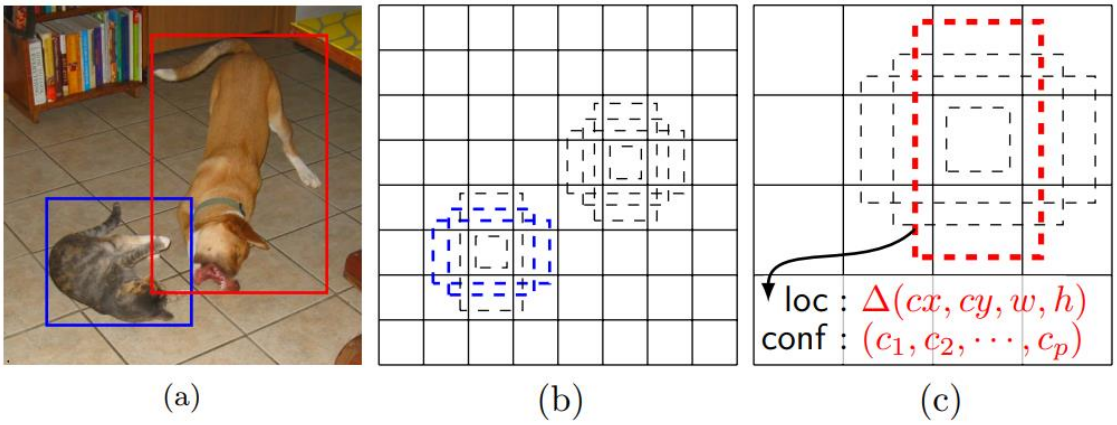

Fig. 2.2 Funcionamiento de un single shot detector [13]

Los SSDs funcionan de la siguiente manera: la red neuronal de múltiples capas de nodos se entrena con imágenes que están marcadas con rectángulos alrededor de los objetos que se desean detectar. Un ejemplo de imagen que se puede utilizar para entrenar, sería la de la **Fig. 2.2 (a)**, donde se puede ver a un perro y a un gato, que son los objetos a detectar encerrados en rectángulos.

Para detectar los objetos, se utilizan cuadrículas (como las de la **Fig 2.2 (b)** y **(c)**), junto con rectángulos de diferentes proporciones, representados con líneas discontinuas en la figura, éstos se colocan en diferentes posiciones y se asocia su localización (coordenadas, altura y anchura) con la probabilidad de que haya alguno de los objetos que se quiere detectar en ese rectángulo.

Durante el entrenamiento se intenta que si un rectángulo contiene un determinado objeto, reconozca correctamente ese objeto. Esto se hace intentando ver qué propiedades para los rectángulos encajan mejor con ese objeto, y así poder utilizar esos rectángulos específicamente para ese tipo de objeto. Se puede observar en la **Fig 2.2 (b)** y **(c)** cómo están marcados los rectángulos que seleccionarían al perro y al gato.

Es importante utilizar varios tamaños de cuadrículas, para poder detectar los objetos aunque aparezcan con un tamaño más grande o pequeño dentro de la imagen.

2.1.2. Extracción de *embeddings* de cada cara

Este segundo paso en el reconocimiento facial, consiste en extraer los *embeddings* de cada cara detectada durante la detección facial.

Los *embeddings* son una serie de números, que forman un vector y que representan las propiedades de una cara, permitiendo así cuantificarla.

Para poder obtener los *embeddings* es necesario primero transformar la cara detectada durante la detección facial, esto significa realizar las rotaciones necesarias para que la cara este correctamente alineada. A continuación, se recorta la imagen para que solo aparezca la cara.

Una vez se tiene una imagen con solo una cara, correctamente alineada, se introduce en una red neuronal de múltiples capas de nodos, también conocida como *Deep neural network*, y se extraen 128 valores que miden distintas propiedades de la cara, los *embeddings*.

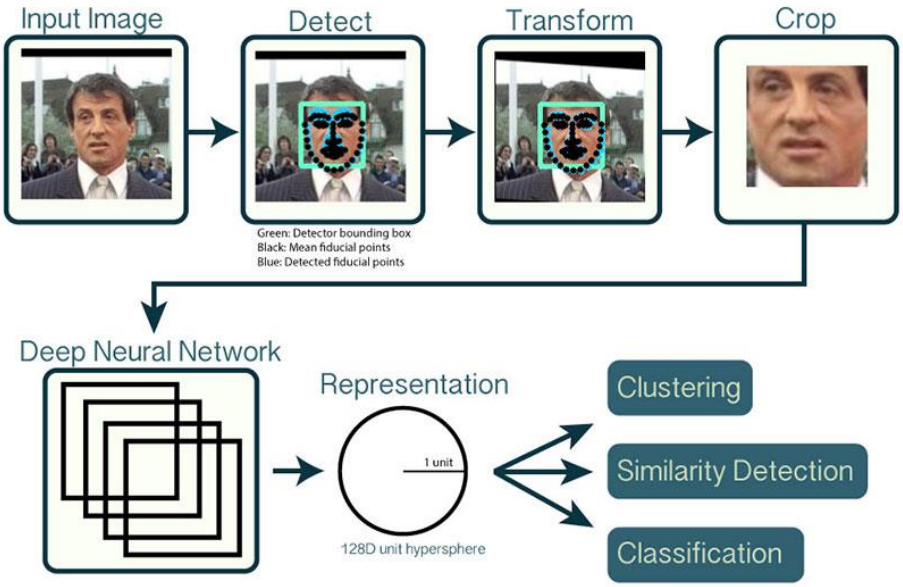

Fig. 2.3 Extracción de *embeddings* de una cara [14]

Una vez se han extraído los *embeddings* de algunas caras se procede al paso más importante que es la utilizar el método *triplet loss*. Este método consiste en utilizar los *embeddings* de tres caras etiquetadas: dos de la misma persona (una será la imagen de prueba) y una de una persona diferente.

Se comparan los *embeddings* de las tres imágenes y se cambian ligeramente los pesos de la red neuronal (*Deep Neural network*), para que la imagen de prueba y la imagen de la misma persona den resultados (*embeddings*) lo más similares posible; mientras que también se cambian los pesos, para que la imagen de prueba y la imagen de la otra persona den resultados lo más diferentes posible. Se muestra un esquema del funcionamiento en la **Fig 2.4**.

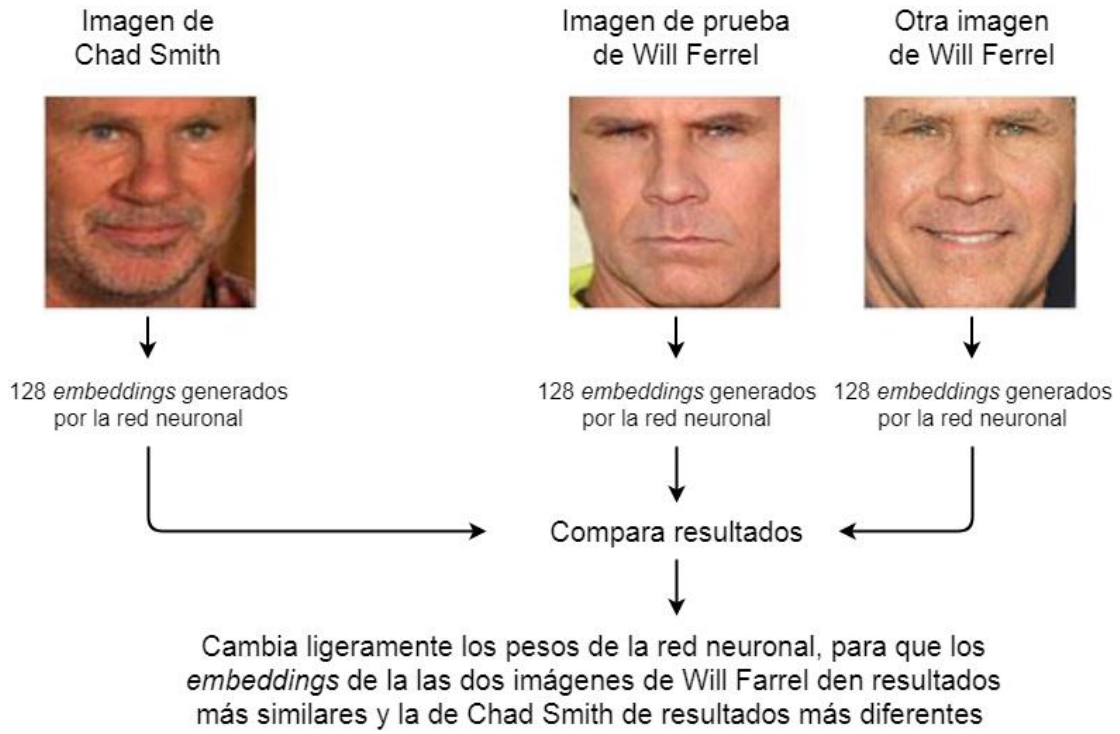

Fig. 2.4 Función triplet loss [15]

2.1.3. Entrenamiento de máquina de vectores de soporte (SVM)

Una vez se han detectado las caras y extraído los *embeddings* de todas las imágenes, se puede proceder a entrenar la máquina de vectores de soporte.

La máquina de vectores de soporte que se utiliza en este proyecto utiliza un *kernel* lineal, ya que los datos que se introducirán tienen 128 dimensiones (128 *embeddings*) por cara.

Se utiliza una máquina de vectores de soporte en vez de otras opciones, como podría ser una red neuronal de múltiples capas de nodos, ya que tienen un coste computacional muy inferior y da resultados correctos cuando hay una cantidad baja de imágenes a clasificar. En caso de que se quiera utilizar una gran cantidad de imágenes, sería más recomendable utilizar una red neuronal de múltiples capas de nodos, ya que en ese tipo de situaciones las máquinas de vectores de soporte no dan tan buenos resultados.

La máquina de vectores de soporte se encarga de hacer *clusters* con los diferentes *embeddings* de todas las imágenes con las que es entrenada, para así posteriormente poder reconocer las caras de los sujetos.

2.1.4. Evaluación de modelo

Ahora que la máquina de vectores de soporte ha sido entrenada, se pueden comenzar a hacer diversas pruebas para evaluar si es adecuada o no para la aplicación para la cual se quiere utilizar.

Para esta evaluación, es necesario seguir los pasos descritos en el apartado 1.6 y obtener: la matriz de confusión, exactitud, precisión y la tasa de verdaderos positivos.

2.2. Implementación del modelo de reconocimiento facial

En este apartado se explicará cómo se ha realizado la implementación del modelo de reconocimiento facial en este proyecto, describiendo las herramientas, códigos y librerías utilizadas.

2.2.1. Configuración del entorno de trabajo

Para implementar este modelo de reconocimiento facial, es necesario realizar una serie de pasos previos que consisten en instalar Python y algunas de sus librerías:

2.2.1.1. Instalar Python

Python es el lenguaje de programación que será utilizado durante esta implementación. Es un lenguaje popular en machine learning, gracias a su sintaxis simple y a la gran variedad de librerías que tiene disponibles.

Para instalar Python, seguir las instrucciones en [13].

Para facilitar la tarea de instalar algunas de las librerías y paquetes de Python que se explicarán a continuación, es recomendable utilizar un instalador de paquetes como puede ser pip [14].

2.2.1.2. Crear un entorno virtual

Una de las funciones más utilizadas en Python es la posibilidad de crear entornos virtuales, estos permiten tener entornos diferentes, con sus propias librerías y dependencias, dependiendo del proyecto en el que se quiera trabajar.

Para crear un entorno virtual hay que seguir los siguientes pasos:

```
py -3 -m venv .venv
.venv\scripts\activate
```

Código 2.1 Crear entorno virtual en Windows

```
python3 -m venv .venv
source .venv/bin/activate
```

Código 2.2 Crear un entorno virtual en Linux

2.2.1.3. Instalar NumPy

NumPy es un paquete de Python que se utiliza para trabajar con vectores y matrices. En este proyecto se utilizará a través de OpenCV, explicado en el siguiente apartado.

Para instalar NumPy seguir las instrucciones en [15].

2.2.1.4. Instalar OpenCV

OpenCV es una de las librerías más utilizadas hoy en día para resolver problemas de visión por computador. Es la que se utilizará durante esta implementación.

Para instalar OpenCV seguir las instrucciones en [16].

Para instalar OpenCV en una Rasbperry Pi, se recomienda la siguiente guía [17], que fue utilizada en este proyecto.

2.2.1.5. Instalar imutils

Imutils es un paquete de Python que incluye funciones que facilitan algunas tareas en el procesado de imágenes, como pueden ser: translación, rotación o redimensionado de imágenes.

Para instalar imutils seguir las instrucciones en [18].

2.2.2. Estructura de carpetas y archivos

La implementación tendrá la siguiente estructura de carpetas y archivos:

```
├── dataset
│   ├── unknown [100 images]
│   └── victor [20 images]
├── face_detection_model
│   ├── deploy.prototxt
│   └── res10_300x300_ssd_iter_140000.caffemodel
├── images [60 images]
├── output
│   ├── embeddings.pickle
│   ├── le.pickle
│   └── recognizer.pickle
├── extract_embeddings.py
├── openface_nn4.small2.v1.t7
├── recognize.py
├── recognize_video.py
└── train_model.py
```

Código 2.3 Estructura de la implementación

En los siguientes apartados se describirán las funciones de las diversas carpetas y archivos.

2.2.2.1. Dataset

Esta carpeta, contiene las fotografías de las diferentes personas que se quieren utilizar para entrenar la máquina de vectores de soporte.

Las fotografías deben estar separadas en carpetas dependiendo de la persona a la cual pertenezcan. En este ejemplo, hay una carpeta con veinte fotografías del sujeto *Víctor* y otra carpeta con cien imágenes que pertenecen a personas desconocidas y por eso recibe el nombre de “Unknown”.

Siempre es necesario que hayan como mínimo dos carpetas distintas, ya que sino la máquina de vectores de soporte no podría ser entrenada. Sí que puede haber más de dos carpetas.

2.2.2.2. Face detection model

Contiene el modelo entrenado previamente de una red neuronal con múltiples capas de nodos, explicado en el apartado 2.1.1. Es proporcionado por OpenCV y sirve para detectar y localizar caras en una imagen.

El archivo *deploy.prototxt* contiene el diseño de la red y el archivo *res10_300x300_ssd_iter_140000.caffemodel* contiene los pesos de la red.

2.2.2.3. openface_nn4.small2.v1.t7

Contiene la red neuronal de múltiples capas de nodos explicada en el apartado 2.1.2. Ya ha sido entrenada previamente y que extrae *embeddings* a partir de las caras detectadas por el modelo del apartado 2.2.2.2.

2.2.2.4. Images

Esta carpeta contiene las imágenes que se utilizarán para realizar pruebas y evaluar la máquina de vectores de soporte una vez entrenada. En este caso contiene 60 imágenes, que son tanto de la persona conocida como de las desconocidas.

2.2.2.5. Output

Estos son los archivos de salida, que se van obteniendo a medida de que se superan los distintos pasos explicados en el apartado 2.1.

- *embeddings.pickle*: una vez los *embeddings* de las caras de la carpeta Dataset son extraidos, se guardan aquí. Por tanto, contiene todos los *embeddings* de las caras con las que la máquina de vectores de soporte es entrenada.
- *le.pickle*: es la codificación de los nombres, es decir, contiene todos los nombres de las personas que la máquina de vectores de soporte es capaz de reconocer.
- *recognizer.pickle*: es el modelo de machine learning, en concreto una SVM, responsable de realizar el reconocimiento facial, explicada con más detalle en el apartado 2.1.3.

2.2.2.6. extract_embeddings.py

Es el script de Python responsable de generar los *embeddings*, que contienen 128 parámetros que describen todas las caras de la carpeta dataset.

Se puede encontrar éste y los siguientes códigos completos en el anexo A.1.

2.2.2.7. train_model.py

Script encargado de entrenar la SVM con los *embeddings* extraídos en el apartado anterior 2.2.2.6.

2.2.2.8. recognize.py

Utiliza la SVM del apartado anterior 2.2.2.7, junto con la capacidad de detectar caras y extraer *embeddings* en las imágenes de la carpeta images, para poder reconocer caras, mostrando también la probabilidad con la que, según el modelo, esa cara corresponde a la persona que cree que corresponde.

2.2.2.9. recognize_video.py

Reconoce caras de las personas con las que se ha entrenado la SVM, dentro de los fotogramas de un vídeo que está siendo grabado en directo con una cámara conectada al dispositivo donde se ejecuta este Script de Python.

2.3. Evaluación del modelo de reconocimiento facial

En este apartado se describirán las diferentes pruebas que se han realizado para la correcta evaluación del modelo de reconocimiento facial descrito en los apartados anteriores.

2.3.1. Descripción de las colecciones de fotografías utilizadas

Para realizar las pruebas que se describirán en los siguientes apartados se utilizaron las siguientes colecciones de fotografías.

2.3.1.1. 15 fotografías Víctor

Está formada por 15 fotografías del sujeto *Víctor*, tomadas con el módulo de Raspberry Pi cámara V2, en diferentes posturas, con diferentes fondos y en diferentes condiciones de iluminación. Se muestran en la **Fig 2.5**.

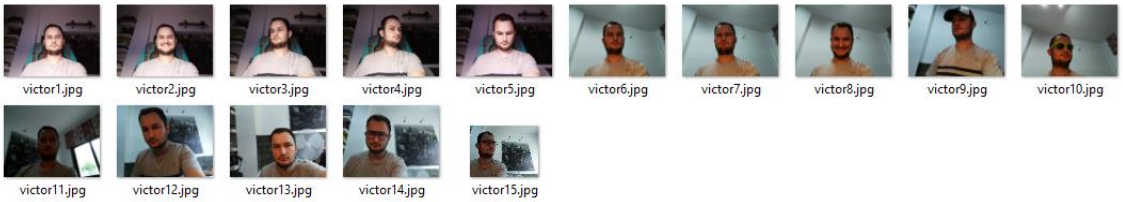

Fig. 2.5 15 fotografías Víctor

2.3.1.2. 30 fotografías Víctor con mejor iluminación

Está formada por 30 fotografías del sujeto *Víctor*, mostradas en la **Fig 2.6**, tomadas con el módulo de Raspberry Pi cámara V2, en diferentes posturas, con el mismo fondo y con las mismas condiciones de iluminación, que son mejores que las de las imágenes descritas en el apartado anterior 2.3.1.1.

Para algunas pruebas, se seleccionaran 15 fotografías de las descritas en este apartado, para poder realizar comparaciones sobre la iluminación con las fotografías del apartado anterior.

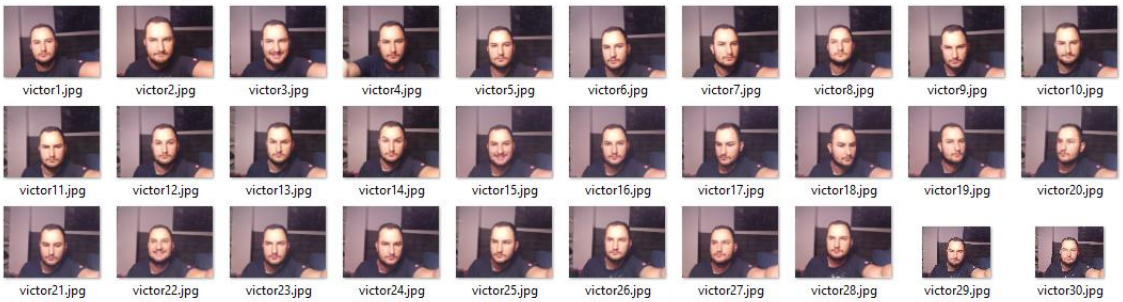

Fig. 2.6 30 fotografías Víctor

2.3.1.3. 30 fotografías Meritxell

Está formada por 30 fotografías del sujeto *Meritxell*, mostradas en la **Fig 2.7**, tomadas con el módulo de Raspberry Pi cámara V2, en diferentes posturas,

con el mismo fondo y con las mismas condiciones de iluminación, que son iguales a las del apartado anterior 2.3.1.2.

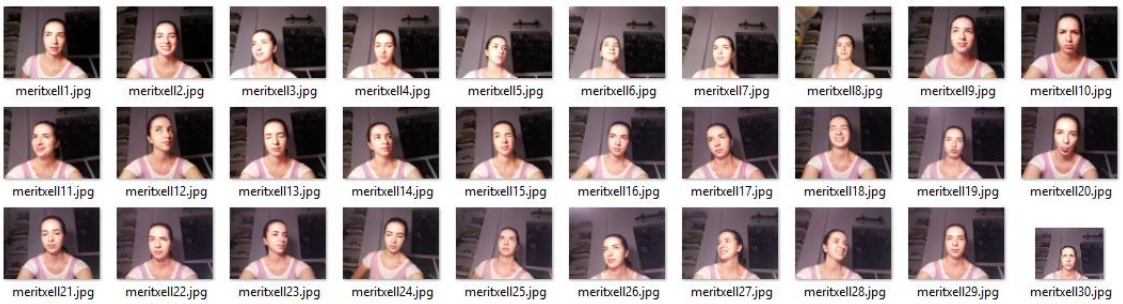

Fig. 2.7 30 fotografías *Meritxell*

2.3.1.4. 150 Fotografías de desconocidos en diferentes posturas

Fue utilizada la base de datos de imágenes del Centro Universitario FEI [19] que contiene imágenes de 200 personas, cada una en 14 posturas diferentes, con diferentes condiciones de iluminación. Se muestran en la **Fig 2.8**.

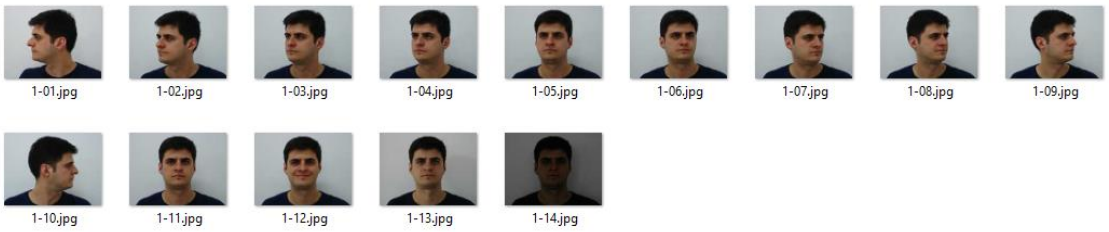

Fig. 2.8 Posturas en las fotografías de desconocidos

De estas imágenes, se seleccionó una de cada persona, variando la postura seleccionada de manera aleatoria, para tener la mayor diversidad de sujetos y posturas para las pruebas. Obteniendo un total de 150 imágenes, mostradas en la **Fig 2.9**.

Fig. 2.9 Algunas de las imágenes seleccionadas en diferentes posturas

2.3.1.5. 150 Fotografías de desconocidos en la misma postura

También fueron seleccionadas 150 fotografías de la base de datos [19] de diferentes personas, pero en la misma postura (mirando a cámara), para poder evaluar si el hecho de que las personas de las fotografías estuvieran en posturas diferentes, o no, afectaba a los resultados obtenidos. Estas fotografías se muestran en la **Fig 2.10**.

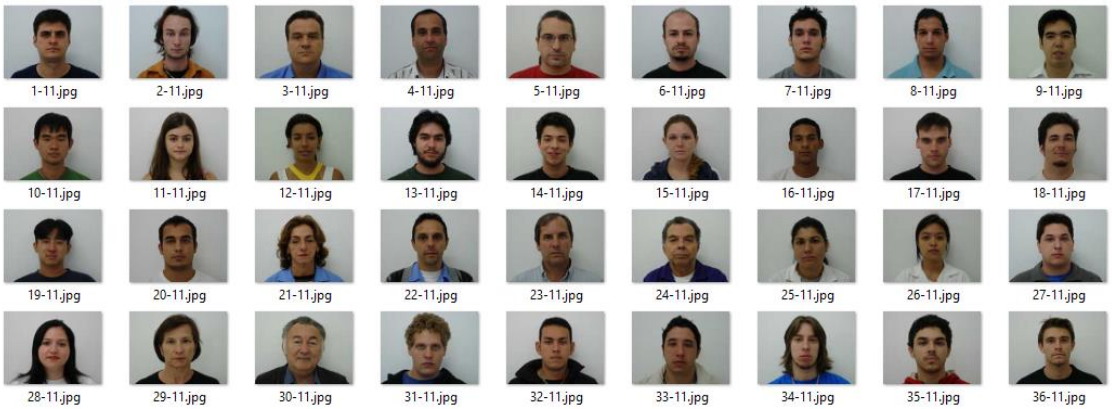

Fig. 2.10 Algunas de las imágenes seleccionadas en la misma postura

2.3.2. Descripción de las pruebas realizadas.

Durante la fase de pruebas del modelo se utilizaron los siguientes conjuntos de imágenes:

- 15 fotografías del sujeto *Víctor* y 150 fotografías de desconocidos en diferentes posturas.

- 15 fotografías del sujeto *Víctor* y 150 fotografías de desconocidos en la misma postura.
- 15 fotografías del sujeto *Víctor* con mejor iluminación y 150 fotografías de desconocidos en diferentes posturas.
- 15 fotografías del sujeto *Víctor* con mejor iluminación y 150 fotografías de desconocidos en la misma postura.
- 30 fotografías del sujeto *Víctor* y 150 fotografías de desconocidos en diferentes posturas.
- 30 fotografías del sujeto *Víctor* y 150 fotografías de desconocidos en la misma postura.
- Variación del número de fotografías del sujeto a identificar.
- Variación del número de fotografías de desconocidos

Se puede encontrar una descripción más detallada tanto de las pruebas como de los resultados en el anexo A.2.

2.3.3. Comparación y conclusiones de las pruebas realizadas

A continuación, se hará una comparación de los diferentes resultados obtenidos, respecto a temas concretos a tener en cuenta cuando se utilice este modelo de reconocimiento facial.

Los parámetros que se utilizarán para poder realizar estas comparaciones correctamente, serán: exactitud, precisión y *recall*. Descritos en el apartado 1.6.2.

2.3.3.1. Comparación de la postura de los desconocidos

Para poder comparar cuales son las consecuencias de tener imágenes de desconocidos en diferentes posturas (como las mostradas en el apartado 2.3.1.4), o en la misma postura (como las mostradas en el apartado 2.3.1.5), se compararán todos los resultados realizados tanto con unas imágenes como con las otras.

Tabla 2.1 Comparación de las posturas de desconocidos

Prueba	Propiedad	Diferentes posturas	Misma postura	Diferencia (diferentes-misma)
15 Víctor + 150 Desconocidos	Exactitud	94,6 % ± 1,8	92,7 % ± 3,15	**1,9 %**
	Precisión	91,7 % ± 14,4	50 % ± 50	**41,7 %**
	Recall	46,7 % ± 23,1	40 % ± 34,6	**6,7 %**
15 Víctor mejor iluminación + 150 Desconocidos	Exactitud	99,4 % ± 1.0	98,8 % ± 2,1	**0,6 %**
	Precisión	94,4 % ± 9,6	93,3 % ± 11,5	**1,1 %**
	Recall	100 % ± 0	93,3 % ± 11,5	**6,7 %**
30 Víctor mejor iluminación + 150 Desconocidos	Exactitud	95,6 % ± 2,4	96,7 % ± 2,9	**-1,1 %**
	Precisión	97,0 % ± 5,2	93,9 % ± 10,5	**3,1 %**
	Recall	78,6 % ± 18,5	86,7 % ± 15,3	**-8,1 %**

Como se puede ver en la **Tabla 2.1**, en la gran mayoría de los casos es mejor tener diferentes posturas, que una sola postura. Para poder evaluar las consecuencias exactas en cada parámetro, se muestra la siguiente tabla:

Tabla 2.2 Media de la comparación de las posturas de desconocidos

	Propiedad	Diferencia media (diferentes-misma)
Media	Exactitud	**0,5 %**
	Precisión	**15,3 %**
	Recall	**1,8 %**

Como se puede apreciar en la **Tabla 2.2**, el parámetro al que más afecta la variación de postura es la precisión, esto significa que si se tienen imágenes de desconocidos en diferentes posturas, será más probable que las predicciones positivas sean correctas. Los otros dos parámetros también son afectados, pero en menor medida, e incluso en una de las pruebas empeoraron en lugar de mejorar.

Se puede suponer que igual que tener más variedad de posturas en sujetos desconocidos mejora la precisión, también mejorará mientras mayor variedad de posturas se tengan en el sujeto a identificar.

2.3.3.2. Comparación de la iluminación

En este apartado se hará una comparación de cómo afecta la iluminación en las imágenes a los resultados obtenidos. Para ello se utilizan las 15 fotografías descritas en el apartado 2.3.1.1, así como 15 de las 30 fotografías del apartado 2.3.1.2, que tienen mejor iluminación que las fotografías del apartado anterior.

Tabla 2.3 Comparación de la iluminación

Prueba	Propiedad	Mejor iluminación	Peor iluminación	Diferencia (mejor-peor)
15 Víctor + 150 Desconocidos diferentes posturas	Exactitud	99,4 % ± 1.0	94,6 % ± 1,8	**4,8 %**
	Precisión	94,4 % ± 9,6	91,7 % ± 14,4	**2,7 %**
	Recall	100 % ± 0	46,7 % ± 23,1	**53,3 %**
15 Víctor + 150 Desconocidos misma postura	Exactitud	98,8 % ± 2,1	92,7 % ± 3,15	**6,1 %**
	Precisión	93,3 % ± 11,5	50 % ± 50	**43,3%**
	Recall	93,3 % ± 11,5	40 % ± 34,6	**53,3 %**

Se muestra en la **Tabla 2.3** como, en todos los casos se obtienen mejores resultados cuando hay mejor iluminación. Igual que en el apartado anterior, se muestra a continuación la media de los parámetros de evaluación:

Tabla 2.4 Media de la comparación de iluminación

	Propiedad	Diferencia media (mejor-peor)
Media	Exactitud	**5,5 %**
	Precisión	**23,0 %**
	Recall	**53,3 %**

Como se puede ver en la **Tabla 2.4**, el parámetro que se ve más afectado es el *Recall*, es decir, cuántos de los casos que contenían la cara del sujeto a identificar fueron identificados. Este es un parámetro muy importante, puesto que el objetivo de este modelo es identificar a este sujeto y siendo un incremento de más del 50%, significa que se debe tener especial consideración en las fotografías del sujeto a identificar.

Los otros dos parámetros también aumentan notablemente al mejorar la iluminación.

2.3.3.3. Variación del número de fotografías del sujeto a identificar

Durante la realización de esta prueba se utilizaron las 30 fotografías descritas en el apartado 2.3.1.2 y se fueron reduciendo de tres en tres, para poder evaluar el efecto de reducir el número de fotografías del sujeto a identificar.

Para poder ver claramente estos resultados se muestra la siguiente figura, con la evolución de cada parámetro en función del número de fotografías:

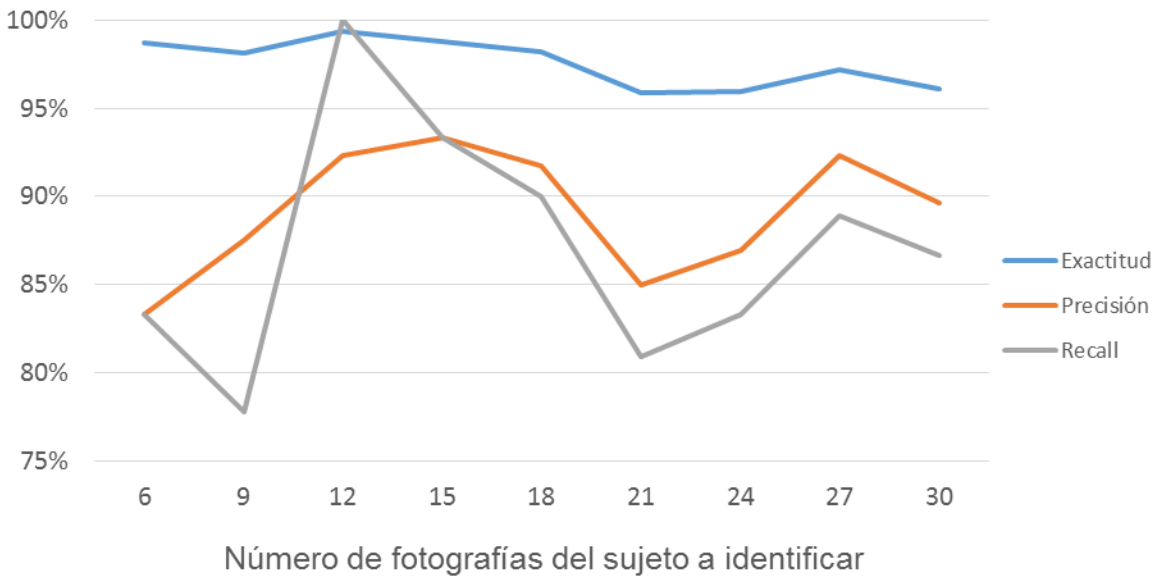

Fig. 2.11 Parámetros en función del número de fotografías del sujeto a identificar

Como se puede observar en la **Fig 2.11**, el valor máximo para los tres parámetros está alrededor de las 12 fotografías, siendo el parámetro más afectado el *recall*. Se ve también como al aumentar el número de fotografías, estos valores tienen tendencia a descender, aunque haya otro pico en los valores alrededor de las 27 fotografías.

Como confusión se puede obtener, que lo óptimo sería tener alrededor de 12 fotografías del sujeto a identificar, pero que si se tiene un número elevado, también es posible obtener buenos resultados.

2.3.3.4. Variación del número de fotografías de desconocidos

Durante la realización de esta prueba se utilizaron las 30 fotografías descritas en el apartado 2.3.1.2, así como las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5, que se fueron reduciendo para poder observar el efecto que esto tenía en los distintos parámetros de evaluación.

Para poder evaluar en detalle la evolución de los parámetros se utilizará la siguiente figura:

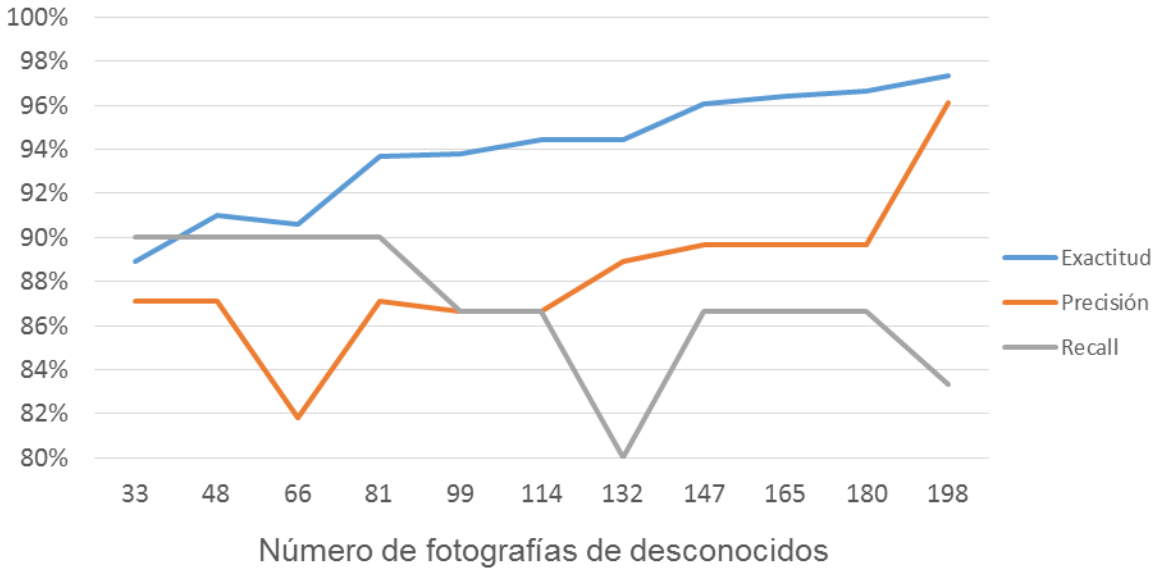

Fig. 2.12 Parámetros en función del número de fotografías de desconocidos

Se puede observar en la **Fig 2.12**, que cuando se incrementa el número de fotografías de desconocidos, se incrementan también notablemente la exactitud y la precisión, mientras que el *recall* oscila con una aparente tendencia a disminuir.

Como conclusión, se debe utilizar el mayor número de fotografías de desconocidos posible, pero teniendo en cuenta que si se obtiene un *recall* muy bajo, puede ser buena idea probar a reducir el número de fotografías de desconocidos.

CAPÍTULO 3. RECONOCIMIENTO FACIAL UTILIZANDO AZURE FACE

En este capítulo se describirá que es Azure Face API y se realizarán pruebas para poder comparar su modelo de reconocimiento facial con el descrito en el CAPÍTULO 2.

3.1. Descripción de Azure Face API

Face API forma parte de Microsoft Azure Cognitive Services y permite detectar, reconocer y analizar caras [20]. En este proyecto se utilizarán dos de las funciones principales que proporciona:

- Un modelo de reconocimiento facial similar al descrito en el CAPÍTULO 2, pero del cual no se pueden conocer exactamente los elementos de los que se compone, ni su funcionamiento exacto.
- Un modelo que analiza la similitud de dos caras. Con dos fotografías y sin ningún entrenamiento adicional, detecta si en las dos fotografías aparece la misma persona y proporciona la probabilidad asociada a esta predicción.

Para su utilización, se realizan una secuencia de peticiones a la API, en las cuales primero se envían imágenes que se asocian a un nombre y a continuación, se realizan peticiones para obtener información sobre las imágenes enviadas anteriormente. De esta manera no es necesario realizar ningún tipo de cálculos ni operaciones en el dispositivo que obtiene los datos, ya que se realizan en los equipos de Microsoft.

Para obtener más información sobre cómo utilizar Azure Face API, es recomendable seguir esta guía [21].

3.2. Descripción de las pruebas de reconocimiento facial

Las pruebas a realizar, serán las mismas que en la primera prueba descrita en el apartado 2.3.2. Consiste en utilizar las 15 fotografías del sujeto *Víctor* con peor iluminación y 150 fotografías de sujetos desconocidos en diferentes posturas, analizando si se identifica correctamente si la imagen pertenece al sujeto *Víctor* o no.

A diferencia de las pruebas del CAPÍTULO 2, ahora no es necesario entrenar la red con fotografías de sujetos desconocidos, ya que el modelo de Azure ya ha sido entrenado con fotografías anteriormente. Por ello, solo se entrenará con las imágenes del sujeto *Víctor*.

Igualmente, se realizarán las pruebas con la misma estructura que en la validación cruzada del CAPÍTULO 2, con la única diferencia de no entrenar con imágenes de desconocidos. Esto permitirá realizar una comparación posterior, ya que las pruebas se han realizado en las condiciones lo más similares posible.

3.3. Resultados de las pruebas de reconocimiento facial

La matriz de confusión obtenida de los resultados de las pruebas es la siguiente:

Tabla 3.1 Matriz de confusión 5 fotografías *Víctor*, 50 fotografías desconocidos en diferentes posturas utilizando Azure Face Api

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	5	0
	Desconocido	0	50

Como se puede ver en la **Tabla 3.1** el modelo de reconocimiento facial ha acertado todas las predicciones, por tanto los resultados de los parámetros de evaluación son los siguientes:

- Exactitud: 100 %
- Precisión: 100 %
- *Recall*: 100 %

Cabe destacar que en algunos casos concretos en los que la cara estaba de perfil o la fotografía era muy oscura, no se ha detectado ninguna cara en la fotografía. En estos casos se ha considerado el resultado como correcto porque la fotografía no era del sujeto a identificar y eran casos realmente complicados.

Algunos ejemplos de casos en los que no ha detectado ninguna cara son los mostrados en la **Fig 3.1**:

Fig. 3.1 Fotografías con caras no detectadas

3.4. Comparación entre Azure Face API y el modelo de reconocimiento facial descrito en el CAPÍTULO 2

En la siguiente tabla se pueden ver los resultados de las pruebas realizadas utilizando las mismas fotografías con los dos modelos:

Tabla 3.2 Comparación de resultados entre Azure y el modelo descrito en el CAPÍTULO 2

	Azure	**Modelo CAPÍTULO 2**
Exactitud	100 % ± 0	94,6 % ± 1,8
Precisión	100 % ± 0	91,7 % ± 14,4
Recall	100 % ± 0	46,7 % ± 23,1

Como se observa en la **Tabla 3.2** el modelo de Azure es superior al modelo de reconocimiento facial descrito en el CAPÍTULO 2 en todos los aspectos, siendo especialmente significativa la diferencia en el *Recall*.

El único ámbito en el que el modelo del CAPÍTULO 2 es superior, es en detectar caras con muy mala iluminación o en las que el sujeto esta de perfil, como las de la **Fig. 3.1**.

3.5. Descripción de las pruebas de similitud entre dos caras

A continuación, se utilizará y evaluará otra de las funcionalidades que ofrece Azure Face API: detectar la similitud entre dos caras, es decir, si dos caras pertenecen o no a la misma persona. Esta herramienta puede resultar especialmente interesante en algunas aplicaciones, ya que no requiere ningún

tipo de entrenamiento, solo las dos imágenes con las caras que se quieren comparar.

Para evaluar el modelo de Azure que proporciona la similitud de dos caras en dos imágenes, se utilizarán las 15 fotografías con peor iluminación del sujeto *Víctor*, descritas en el apartado 2.3.1.1.

Las pruebas consistirán en utilizar cada una de las 15 fotografías y compararlas con cada una de las otras 14, evaluando así si el modelo es capaz de reconocer que es la misma persona en las dos fotografías, utilizando todas las combinaciones posibles con las fotografías que se disponen.

Para clarificar como se realizan estas pruebas, se puede observar la **Fig 3.2** con algunos de los posibles casos:

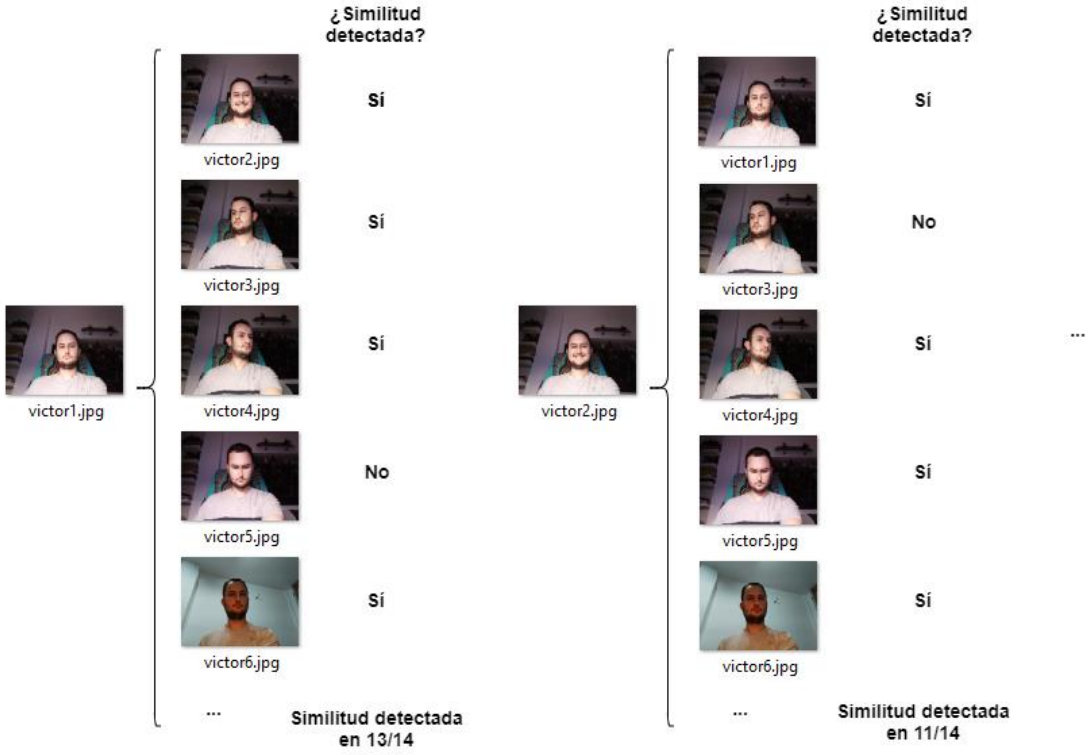

Fig. 3.2 Prueba de similitud en caras

3.5.1. Resultados de las pruebas de similitud entre dos caras

Los resultados de las pruebas de similitud de dos caras son los siguientes:

Fig. 3.3 Resultados en las pruebas de similitud

En la **Fig. 3.3** se observa como en general entre todas las fotografías se encuentran similitudes frecuentemente. El peor caso sería la fotografía 9 en la que el sujeto aparece con una gorra que le tapa parcialmente la cara, aun así se encuentra similitud con 2/3 de las otras fotografías.

Las demás fotografías presentan buenos resultados, destacando positivamente algunos casos como el de la fotografía 10, que pese a llevar el sujeto gafas de sol, es identificado en 12 de las otras 14 fotografías. Es destacable también el caso de la fotografía 11, que tiene poca iluminación al estar a contraluz, pero es identificada en todas las 14 fotografías correctamente.

Como conclusión, este modelo no es tan bueno como el de reconocimiento facial de Azure, pero para no tener que realizar ningún entrenamiento adicional y utilizando solo dos fotografías, da resultados realmente destacables.

CAPÍTULO 4. ANÁLISIS DEL CONSUMO Y TIEMPO

En este cuarto capítulo se realizará un análisis del consumo y del tiempo empleado por los modelos de reconocimiento facial vistos en capítulos anteriores.

Se utilizarán dos modos de computación distintos [25]: *edge* y *cloud*, los cuales se analizarán y se compararán sus características.

La computación *edge* consiste en que el dispositivo que obtiene los datos, realice también el procesado de estos datos, sin necesidad de estar conectado a ningún otro dispositivo para ello. Normalmente, se trata de dispositivos sencillos, sin mucha potencia de cálculo ni mucha memoria. Tienes las ventajas de una solución distribuida (descentralizada) porque las tareas se reparten entre muchos dispositivos.

En el caso de este proyecto, consiste en realizar los cálculos y las operaciones necesarias del modelo de reconocimiento facial, en el mismo dispositivo que obtiene los datos.

Para realizar las pruebas de computación *edge* se utilizará el modelo de reconocimiento facial descrito en el CAPÍTULO 2 en una Raspberry Pi.

La computación *cloud* consiste en que los dispositivos que obtienen los datos, los envíen a otro dispositivo que realice el procesado de estos datos y que este último, a su vez, envíe los resultados al dispositivo que corresponda. Es una solución centralizada, donde los cálculos son realizados en una máquina de gran potencia, pero que está alejada de los puntos donde se recogen los datos.

En el caso de este proyecto, un dispositivo obtendrá los datos, es decir, las imágenes de las personas, las enviará a Azure y recibirá una respuesta con información sobre la imagen enviada.

Para realizar las pruebas de computación *cloud* se utilizará la API de *Azure Face* descrita en el CAPÍTULO 3, a la que se accede desde una Raspberry Pi y también desde un ordenador portátil (ver 4.1.2) para alguna de las pruebas.

4.1. Descripción del hardware utilizado

En este apartado se describirá el hardware utilizado en este capítulo.

4.1.1. Raspberry Pi

Raspberry Pi [22] es un ordenador de bajo coste y reducidas dimensiones, que permite conectar un teclado, ratón y pantalla para poder ser utilizado como un ordenador convencional. Tiene la particularidad de que gracias a su facilidad para conectar sensores y a su bajo consumo, puede ser utilizado en múltiples aplicaciones en donde no se podría utilizar un ordenador convencional, especialmente en aplicaciones IoT (Internet de las cosas).

El sistema operativo que se utilizará en este proyecto será Raspbian [23], un sistema operativo gratuito basado en Debian, que está especialmente diseñado para la Raspberry Pi. Para poder obtener e instalar este sistema operativo se recomienda el siguiente sitio web [24].

Existen diversos modelos de Raspberry Pi, que pueden ser utilizados para diferentes aplicaciones. En este proyecto se utilizará el modelo Raspberry Pi 3 Model B V1.2.

4.1.2. Ordenador portátil

En las pruebas de este capítulo también se utilizará un ordenador portátil, exclusivamente para comparar algunos escenarios con la Raspberry Pi.

El ordenador portátil utilizado cuenta con un procesador i3-5005u y 4GB de memoria RAM.

4.2. Descripción de las pruebas realizadas

En este apartado se describirán las diferentes pruebas de consumo y tiempo realizadas. Todas las pruebas han sido realizadas tres veces y los resultados que se mostrarán en el siguiente apartado 4.3, serán la media de las tres pruebas.

Todas pruebas consisten en entrenar el modelo en cuestión con imágenes de dos sujetos:

- 10 imágenes del sujeto *Víctor* descritas en el apartado 2.3.1.2
- 10 imágenes del sujeto *Meritxell* descritas en el apartado 2.3.1.3

A continuación, se utiliza el modelo para reconocer a cuál de los dos sujetos pertenecen las siguientes imágenes:

- 5 imágenes del sujeto *Víctor* descritas en el apartado 2.3.1.2
- 5 imágenes del sujeto *Meritxell* descritas en el apartado 2.3.1.3

Esta fase de reconocimiento es en la que se centran las pruebas realizadas en este capítulo, ya que en una aplicación real, el entrenamiento se debería haber realizado previamente en otro dispositivo.

Las pruebas a realizar son las siguientes:

- **Prueba *edge* en Raspberry Pi:**

  Se realizan las pruebas de reconocimiento en la Raspberry Pi, utilizando el modelo de reconocimiento descrito en el CAPÍTULO 2.

- **Prueba *cloud* en Raspberry Pi con conexión Wi-Fi:**
  Se realizan las pruebas de reconocimiento desde la Raspberry Pi, utilizando el modelo de reconocimiento de Azure, descrito en el CAPÍTULO 3.

  La conexión a internet se establece mediante Wi-Fi.

- **Prueba *cloud* en Raspberry Pi con conexión mediante cable:**
  Se realizan las pruebas de reconocimiento en la Raspberry Pi, utilizando el modelo de reconocimiento de Azure, descrito en el CAPÍTULO 3.

  La Rapsberry Pi solo captura la imagen y la envía a Azure para que haga el reconocimiento.

  La conexión a internet se establece mediante cable, utilizando el mismo router que en la prueba anterior.

- **Prueba *cloud* en un ordenador portàtil:**
  Se realizan las pruebas de reconocimiento en un ordenador portátil, utilizando el modelo de reconocimiento de Azure, descrito en el CAPÍTULO 3.

  La conexión a internet se realiza mediante cable. Esta prueba se realizará en dos escenarios diferentes, con diferentes velocidades de conexión.

4.3. Resultados y comparaciones

A continuación, se mostrarán y compararán los resultados de las diversas pruebas realizadas.

4.3.1. Resultados de consumo en Raspberry Pi

Las gráficas que muestran los consumos en las diferentes pruebas realizadas en la Raspberry Pi son las siguientes:

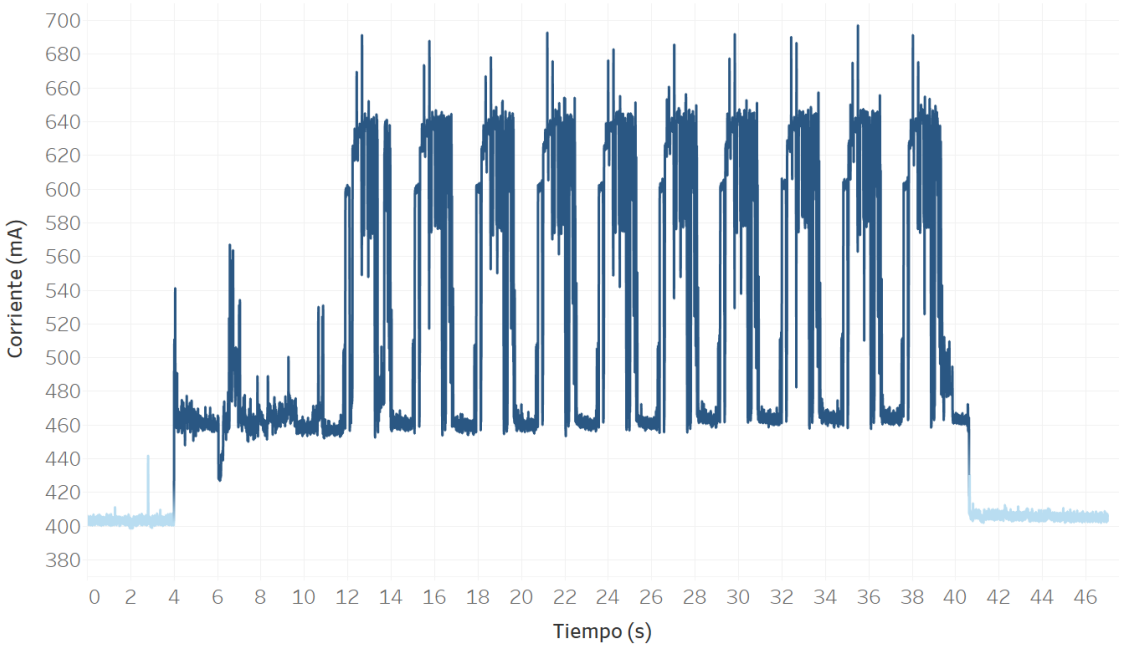

Fig. 4.1 Resultados de consumo de corriente *edge*

En la **Fig. 4.1** se puede observar el consumo de corriente para el caso *edge*. La parte en azul oscuro corresponde al periodo de tiempo desde que se ejecuta el programa, hasta que se obtiene y enseña en pantalla el resultado de la última foto.

El tiempo inicial que aparece aproximadamente del segundo 4 al 12, corresponde al periodo de tiempo en el que se cargan las librerías y los diferentes modelos: detección facial, extractor de *embeddings* y reconocimiento facial. El modelo de reconocimiento facial es el que tarda más en cargar, tardando unos 3,8 segundos.

Después del segundo 12, se procede al reconocimiento facial de las diferentes imágenes, habiendo diez picos de corriente, ya que se hace para diez imágenes diferentes.

El consumo de corriente en el caso *edge* de la **Fig. 4.1** es elevado. Tiene un largo tiempo de preparación inicial antes de realizar el reconocimiento facial en las imágenes. También resulta interesante que hasta que no acaba todo el reconocimiento facial, no se regresa al consumo de reposo.

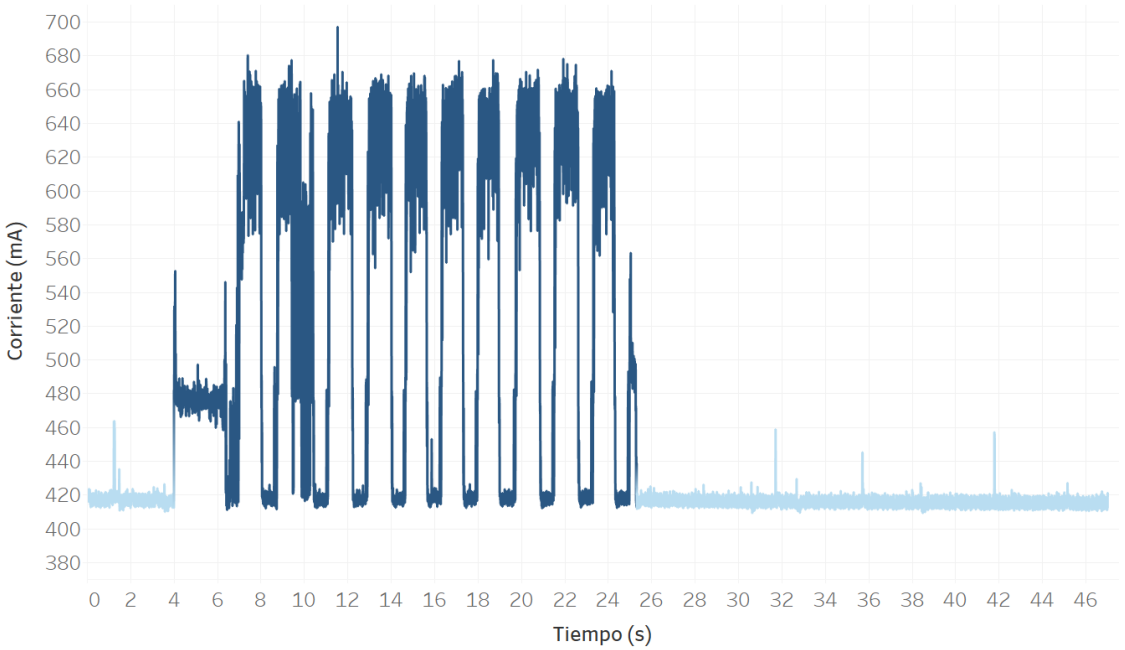

Fig. 4.2 Resultados de consumo de corriente *cloud* con conexión Wi-Fi

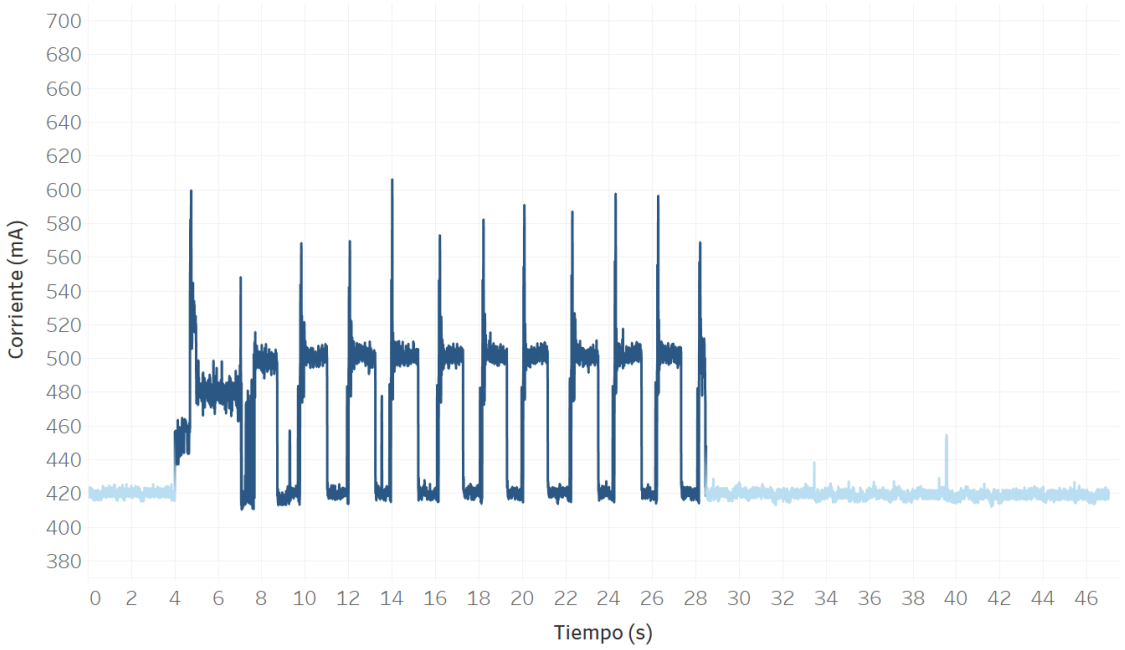

Fig. 4.3 Resultados de consumo de corriente *cloud* con conexión mediante cable

Tal y como se muestra en las figuras **Fig 4.2** y **Fig 4.3** el consumo ofrecido por la opción basada en *cloud* es notablemente menor, así como más corto el tiempo de preparación inicial, que en este caso solo corresponde a la

autenticación. Además, se reduce el consumo al de reposo entre el reconocimiento facial en una imagen y en la siguiente.

El consumo en el caso que se utiliza Wi-Fi es más elevado que en el caso en que se utiliza cable. En este último, el consumo es en general bajo y solo tiene picos de consumo puntuales al inicio de cada reconocimiento.

En la siguiente tabla se puede ver el consumo medio de corriente y potencia, así como la energía consumida durante reconocimiento facial.

Tabla 4.1 Comparación del consumo

	Conexión	**Corriente media**	**Potencia media**	**Energía**
Edge	No aplica	544 mA ± 9	2,72 W ± 0,04	27,73 mW·h ± 0,50
Cloud	Wi-Fi	535 mA ± 5	2,67 W ± 0,02	16,71 mW·h ± 1,24
	Cable	469 mA ± 2	2,35 W ± 0,01	15,17 mW·h ± 0,82

Como se puede apreciar en la **Tabla 4.1**, el menor valor de consumo corriente y potencia media se produce para el caso *cloud* con conexión mediante cable, siendo los otros dos casos similares, aunque inferior el caso *cloud* con Wi-Fi que el *edge*.

La diferencia más significativa, se observa en la energía, donde se obtienen valores similares para los casos *cloud*, pero un valor muy superior para el caso *edge*, esto es debido a que, aunque los consumos de corriente sean relativamente similares en los tres casos, el tiempo utilizado en el caso *edge* es muy superior y esto afecta en gran medida a la energía consumida.

En realidad, el consumo total no se puede comparar, porque no se conoce el consumo de los servidores remotos para el modelo *cloud*. Pero sí se puede asumir que, si las máquinas que utilizan están optimizadas para realizar estos cálculos de forma masiva y el sistema *cloud* está bien dimensionado para su carga, será más eficiente.

4.3.2. Resultados de tiempo en Raspberry Pi

Los resultados de las medidas de tiempo corresponden a la parte de color azul oscuro de las figuras **Fig 4.1**, **Fig 4.2** y **Fig. 4.3**.

Tabla 4.2 Comparación del tiempo

	Conexión	Velocidad Subida	Velocidad Bajada	Tiempo
Edge	No aplica	No aplica	No aplica	36,7 s ± 0,1
Cloud	Wi-Fi	28,5 Mb/s ± 1,6	40,2 Mb/s ± 0,2	22,5 s ± 1,8
	Cable	71,7 Mb/s ± 1,0	84,6 Mb/s ± 3,3	23,3 s ± 1,3

En la **Tabla 4.2** se puede ver como el tiempo para los dos casos *cloud* es similar y muy inferior al tiempo del caso *edge*, esto tiene una gran repercusión en el consumo de energía como ya se explicó en el apartado 4.3.1.

También es importante ver que, aunque la velocidad de la conexión a internet tenga un valor muy inferior en el caso Wi-Fi que en el caso de conexión con cable, el tiempo empleado es superior en el caso con cable. Esto probablemente sea un caso puntual de estas pruebas en concreto y en general no sea superior el tiempo con cable al tiempo con Wi-Fi, pero es interesante ver que el tiempo da resultados similares independientemente de la velocidad de la conexión. Aparentemente, el retardo debido a la transmisión hacia Azure, no es muy relevante en comparación al tiempo de respuesta de la plataforma.

4.3.3. Resultados de tiempo para *cloud* en Raspberry Pi y ordenador portátil

En este apartado se hará una comparación de los tiempos para el escenario *cloud* con cable, entre la Raspberry Pi y el ordenador portátil.

Los tiempos han sido medidos de forma distinta a los apartados anteriores. Se ha utilizado la librería time de Python [26], que permite medir el tiempo dentro del código y por tanto, facilita la comparación entre la Raspberry Pi y el ordenador portátil.

Por este motivo, los tiempos resultan inferiores a los que se muestran en el apartado anterior 4.3.2, pero es solo por la forma utilizada para su medición.

Tabla 4.3 Comparación del tiempo entre Raspberry Pi y el ordenador portátil

	Velocidad Subida	Velocidad Bajada	Tiempo
Raspberry Pi	71,7 Mb/s ± 1,0	84,6 Mb/s ± 3,3	20,5 s ± 1,1
Ordenador Portátil	89,5 Mb/s ± 0,4	90,4 Mb/s ± 0,1	11,6 s ± 0,5

Como se puede ver en la **Tabla 4.3**, la velocidad de conexión es ligeramente superior en el ordenador portátil. A pesar de esta pequeña diferencia en velocidad de conexión, el tiempo de respuesta se reduce a la mitad, gracias a la mayor capacidad de proceso del portátil.

4.3.4. Resultados de tiempo para *cloud* en el ordenador portátil con una velocidad inferior

Para entender mejor el efecto de la velocidad de conexión sobre el tiempo de respuesta de la solución *cloud*, en este apartado se realizará una prueba con el ordenador portátil y con una conexión a internet que tiene una velocidad más reducida que la utilizada en las pruebas anteriores. El resto de condiciones son exactamente las mismas que en aparado anterior 4.3.3.

Tabla 4.4 Comparación de tiempo *cloud* en el ordenador portátil con diferentes velocidades

	Velocidad Subida	**Velocidad Bajada**	**Tiempo**
Resultado anterior	89,5 Mb/s ± 0,4	90,4 Mb/s ± 0,1	11,6 s ± 0,5
Velocidad inferior	2,9 Mb/s ± 0,2	14,5 Mb/s ± 0,9	575,5 s ± 13,9

Por motivos comparativos en la **Tabla 4.4** se han mostrado también los resultados con alta velocidad de conexión obtenidos anteriormente (4.3.3).

Como se puede observar, al contrario de lo visto en el apartado 4.3.2, la velocidad sí que importa, especialmente cuando se reduce a valores muy pequeños. El resultado es que produce un gran incremento del tiempo necesario para realizar el reconocimiento facial, si se utiliza computación *cloud.*

4.3.5. Conclusiones

Respecto al consumo medio de corriente y potencia, hemos comprobado que es similar en cualquiera de los tres casos, aunque siempre inferior para los casos *cloud* respecto al caso *edge*, esto se debe a que los cálculos no se realizan localmente, sino en una máquina remota. Donde sí se muestran grandes cambios es en lo que respecta a la energía, debido al gran tiempo necesario para realizar el reconocimiento facial localmente en la Raspberry Pi.

Respecto al tiempo, éste es muy superior en el caso *edge*. Entre los casos *cloud*, se obtiene un tiempo similar cuando se utiliza el mismo dispositivo, de manera casi independiente a la velocidad que se tenga, siempre y cuando se

tengan velocidades relativamente altas. A medida que la velocidad se reduce a valores muy bajos, empieza a producir un gran incremento del tiempo que hace que la solución se convierta en la opción más rápida.

Destaca la comparación entre Raspberry Pi y el ordenador portátil, ya que, aunque tengan una conexión con velocidades similares, hay una gran diferencia de tiempo, por lo que aunque se utilice un modelo *cloud*, el hardware utilizado sigue teniendo gran importancia.

Como conclusión final, el principal caso que destaca, en donde puede ser más interesante utilizar la versión *edge* que la *cloud*, es cuando se deba colocar el dispositivo en un lugar con una conexión a internet de velocidad reducida y el tiempo sea un factor de gran importancia. Para todos los demás escenarios vistos en este proyecto, sería más interesante utilizar la versión *cloud*.

CAPÍTULO 5. DESARROLLO DE UNA HERRAMIENTA PARA PREDECIR EL TIEMPO DE ESPERA EN COLA EN UN AEROPUERTO

Este capítulo describe en detalle el desarrollo de una herramienta de machine learning, que tiene como objetivo predecir el tiempo de espera en cola en un aeropuerto. Los principales contenidos a tratar serán: el diseño del sistema de obtención de datos en un entorno real, la obtención y procesado de los datos para entrenar el algoritmo, la selección e implementación de diferentes algoritmos y finalmente, la evaluación y comparación de estos algoritmos.

5.1. Diseño del sistema de obtención de datos

Para poder implementar la herramienta de machine learning que intentará predecir el tiempo de espera en cola, es necesario primero obtener los datos que se introducirán en el algoritmo para realizar esta predicción. Para ello se ha diseñado un sistema que permite obtener los datos en un entorno real.

El sistema cuenta con dos dispositivos Raspberry Pi (ver apartado 4.1.1), equipados cada uno con una cámara y un sensor de ultra sonidos. Una de las Raspberry Pi se colocará a la entrada de la cola y la otra a la salida, sin necesidad de tener comunicación entre ellos. El objetivo de estos dispositivos será contar el número de personas que entran y salen de la cola y almacenarlos en un fichero, para a continuación enviarlo a otro dispositivo que realizará el procesado de estos datos. El diseño del sistema se puede ver en la **Fig. 5.1**.

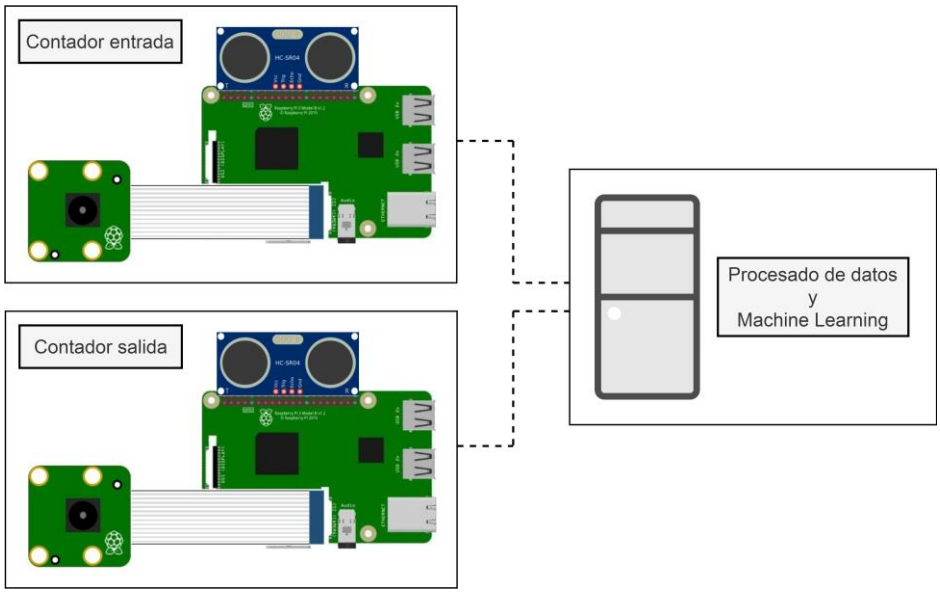

Fig. 5.1 Diseño del sistema de obtención de datos

Para realizar el conteo de personas, tanto a la entrada como a la salida de la cola, el funcionamiento será el mismo. Se deberá colocar el sensor de ultrasonidos de manera que si una persona entra a la cola, tenga que pasar por delante y el sensor pueda detectar su presencia. A continuación, la cámara detectará si lo que ha activado el sensor es una persona, utilizando las técnicas de detección facial explicadas en el apartado 2.1.1. Si se detecta que es una persona, se añadirá información a un fichero, indicando si es una entrada o una salida y el instante de tiempo en el que ha ocurrido. Después de un tiempo determinado, se podrán enviar los datos recopilados por las Raspberry Pi al ordenador que realizará el procesado de estos datos y el entrenamiento del algoritmo de machine learning.

5.2. Obtención de los datos de entrenamiento

En este trabajo, no ha sido posible realizar pruebas en entornos reales y se ha optado por buscar bases de datos con información relevante. Por ello, los datos que se utilizarán son proporcionados por la Oficina de Aduanas y Protección Fronteriza de los Estados Unidos, que ha hecho pública una base de datos [27] que muestra los tiempos de espera en aeropuertos, a partir de 2008.

Los datos que se utilizarán de esta base de datos, proporcionan información agregada para cada hora del día, pero para poder poner a prueba el sistema, es necesario generar llegadas que imiten el comportamiento en tiempo real. Por eso, se ha desarrollado un código que transforma los datos públicos de los aeropuertos en una secuencia temporal, que asume que las llegadas de los viajeros siguen una distribución de Poisson con una tasa de llegadas constante en cada hora, calculada a partir de los datos disponibles.

Los tiempos de espera mostrados en esta base de datos son los correspondientes al control de pasaporte. Se pueden obtener para todos los principales aeropuertos y están separados por terminal. Muestran hora por hora diversos parámetros, de los cuales se utilizarán en este proyecto los siguientes:

- Fecha.
- Hora.
- Tiempo de espera medio.
- Número de pasajeros totales por hora.
- Número de pasajeros excluidos por hora.
- Número de mostradores abiertos en esa hora.

Los datos escogidos serán los del aeropuerto de Los Angeles, LAX, en concreto para la terminal internacional Tom Bradley. Esta selección se debe a que es un aeropuerto y en concreto una terminal grande y con tiempos de espera considerables, que permitirán posteriormente obtener resultados interesantes y relevantes.

La franja temporal de los datos seleccionados es de todo el año 2019, esto nos permitirá tener datos en diferentes épocas del año para el entrenamiento del

algoritmo. Así podrá ajustar su predicción dependiendo de la época, que seguramente hará variar el comportamiento de las colas significativamente.

5.3. Procesado de los datos

Para poder introducir los datos en el algoritmo de machine learning, se debe realizar un procesado previo que adapte los datos al formato correcto para la entrada al algoritmo.

El procesado se dividirá en dos partes: pasar del formato de salida de la base de datos, al formato de salida que tendrían estos datos si se hubiesen obtenido utilizando dos Raspberry Pi, siguiendo el diseño del apartado 5.1; y posteriormente, pasar de este formato al formato de entrada al algoritmo.

El script que pasará del formato de salida de la base de datos al formato de salida de las Raspberry Pi, se llamará: **formatoRPi.py**

El script que pasará del formato de las Raspberry Pi, al formato de entrada al algoritmo de machine learning, se llamará: **formatoML.py**

El escenario completo será como el mostrado en la figura **Fig 5.2**.

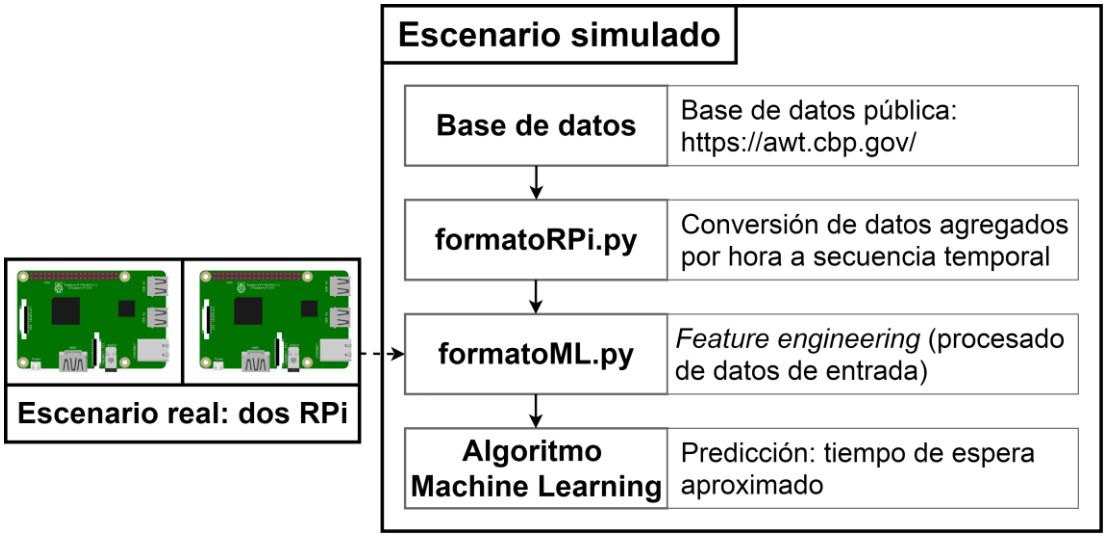

Fig. 5.2 Escenario para el procesado de datos de entrenamiento

Estos scripts, serán explicados en detalle a continuación y el código completo se puede encontrar en el anexo B.1.

5.3.1. Funcionamiento formatoRPi.py

Con los datos obtenidos, especificados en el apartado 5.2, se deben generar datos como los que se obtendrían a la salida de las Raspberry Pi en el sistema diseñado en el apartado 5.1. Estos datos de salida, serán lo más simples posible en lo que respecta al procesamiento, para intentar minimizar la carga de trabajo en estos dispositivos.

El formato de los datos para las entradas a la cola será el siguiente:

"arrival"; año; mes; día; hora; minutos; segundos; número de mostradores

Siendo los datos de: año, mes, día, hora, minutos y segundos, los correspondientes al instante en que una persona entra en la cola. El número de mostradores será el número de mostradores abiertos en el momento que entra la persona a la cola.

El formato de los datos para las salidas de la cola será el siguiente:

"departure"; año; mes; día; hora; minutos; segundos

Siendo los datos de: año, mes, día, hora, minutos y segundos, los correspondientes al instante en que una persona sale de la cola.

A continuación, se describirán las cuatro funciones que componen este script.

5.3.1.1. set_arrivals

Esta función lee cada una de las filas de los datos sin procesar en formato *.csv* y obtiene, calcula y almacena los datos que se utilizarán posteriormente:

- *time*: objeto *datetime* que contiene la fecha y hora a la que hace referencia cada fila.
- *avgWaitTime*: tiempo medio de espera para esa hora, obtenido directamente de cada fila.
- *nArrivals*: número de entradas durante esa hora. Calculado restando los pasajeros excluidos de la cola del total de pasajeros.
- *booths*: número de mostradores abiertos durante esa hora.
- *lamb*: representa la tasa de entradas (entradas por unidad de tiempo). Se utilizará como unidad entradas/segundo, por lo que para calcularla se deberá dividir el número de entradas entre 3600.

Utilizando la tasa de entradas, se generan tiempos de entrada, utilizando la función *hour_Poisson_arrivals*, descrita en el siguiente apartado 5.3.1.2.

Una vez generados los tiempos de entrada, se almacenan en una *list* llamada *arrivals*, junto con el tiempo medio de espera en cola y el número de mostradores abiertos, para su posterior utilización en otras funciones.

5.3.1.2. hour_Poisson_arrivals

La función *hour_Poisson_arrivals*, genera una serie de entradas para una hora concreta utilizando la tasa de entradas.

Para ello se utiliza la librería *random* de Python, en concreto la función *expovariate*, que genera un número aleatorio entre cero e infinito, siguiendo una distribución exponencial que tendrá como media uno entre el valor que se introduzca como parámetro. Con esta función *expovariate* se generarán tiempos aleatorios para las entradas, utilizando como parámetro la tasa de entradas obtenida anteriormente en la función *set_arrivals*.

El instante en el que ocurre la primera entrada de cada hora, será calculado utilizando *expovariate* para generar un tiempo aleatorio que será sumado al inicio de la hora en cuestión (p. ej. 15:00:00 + tiempo aleatorio de 1 min y 12 segundos). Una vez calculada la primera entrada, se calculará otro tiempo aleatorio, pero esta vez se sumará al tiempo de la entrada anterior (p. ej. Para la segunda entrada 15:01:12 + otro tiempo aleatorio) este tiempo se convertirá en siguiente instante en que hay una entrada en la cola. Así sucesivamente con el resto de entradas.

Como el parámetro que se utiliza en la función *expovariate* siempre es la tasa de entradas, el número de entradas en esa hora obtenido de la base de datos coincidirá, en medida, con el número de entradas que ha generado esta función.
Todos estos tiempos de entrada se irán almacenando y se pasarán a la función *set_arrivals*, para que los almacene en la *list arrivals*.

5.3.1.3. set_departures

Esta función genera las salidas de la cola a partir de las entradas, obtenidas anteriormente y del tiempo medio de espera, obtenido directamente de la base de datos para cada hora.

El funcionamiento es el siguiente: se itera para todas las entradas a la cola, que se habían guardado previamente en la *list arrivals*. Para cada entrada, se genera un tiempo aleatorio con la función *expovariate*, utilizando como parámetro uno entre el tiempo medio de espera, con unidades de salidas por segundo. Una vez se tiene este tiempo aleatorio se suma al momento de entrada a la cola y se obtiene el instante de salida.

Una vez generada cada salida, se añade a la *list deparutes*. Esto se repite hasta tenerlas todas almacenadas, una por cada entrada.

Como el parámetro que se ha utilizado en la función *expovariate* es el tiempo medio de espera, si se calculase el tiempo medio de espera para cada hora de la *list departures*, sería muy similar al tiempo medio de espera obtenido de la base de datos. Habría ligeras diferencias, debido a la aleatoriedad introducida.

5.3.1.4. write_csv_data

Finalmente, la función *write_csv_data* escribe los datos generados para las entradas y las salidas en dos ficheros *.csv*, con el formato especificado anteriormente en el apartado 5.3.1.

Estos datos se utilizarán posteriormente en el script *formatoML.py*.

5.3.2. Funcionamiento formatoML.py

Este código realiza el procesado de los datos en el formato adecuado para el entrenamiento del algoritmo de machine learning.

Los datos que se deberán tener a la salida del script y que serán, por tanto, los datos de entrada (*input features*) del algoritmo de machine learning, son los siguientes:

- **Día de la semana**: se considera que es un parámetro relevante ya que el tráfico aéreo varía dependiendo del día de la semana y especialmente, no será al mismo en los días laborables que en fin de semana.

  Se utilizará un *one-hot vector*, que consistirá en 7 *features* diferentes, siendo para cada día seis iguales a cero y una igual a uno, que representará el día de la semana:
  1;0;0;0;0;0;0 Lunes
  0;1;0;0;0;0;0 Martes …

- **Día del año**: consideramos que es un dato de entrada relevante porque el tráfico aéreo puede tener variaciones regulares de acuerdo al día del año. Especialmente en épocas vacacionales como pueden ser los meses de verano, festivos como Navidad, etc.

  Para su cálculo, se obtendrá el día del año, que será un número entre 1 y 366. Como introducir directamente el número podría ser complicado de cuantificar para el algoritmo (p. ej. el 31 de diciembre es 366 veces más grande que el 1 de enero y solo hay un día de diferencia), se deberán transformar utilizando una función cíclica para que los días cercanos tengan valores similares.

  El método escogido, será pasar de un valor entre 1 y 366 a un número entre 0 y 2 π, a continuación calcular el seno y el coseno, que serán los valores que se introducirán en el algoritmo. De esta manera, el 1 de enero y el 31 de diciembre tendrán valores muy similares.

- **Minuto del día**: es relevante porque el tráfico puede variar según la hora del día, por ejemplo, siendo especialmente superior durante las primeras horas de la mañana en días laborables.

Para su cálculo, se obtendrá el minuto del día, que será un valor entre 0 y 1440. De manera similar a como ocurre con el día del año, es necesario transformarlo (p. ej. si no se realizase ninguna transformación, las 00:00 serían 0 y las 23:59 serían 1440 aunque solo haya un minuto de diferencia).

Para transformarlo, pasará de un valor entre 0 y 1440 a un número entre 0 y 2 π, se calculará el seno y el coseno, y estos serán los valores a introducir en el algoritmo de machine learning.

- **Tiempo de espera de la última persona en salir de la cola**: es el tiempo que ha estado en cola la última persona que ha salido. Las unidades son de minutos. Es relevante porque puede ser un buen indicativo el tiempo que tardaría alguien que entrase ahora a la cola.

- **Número de personas en cola**: en general, a medida que más personas haya en la cola, más tiempo de espera habrá.

- **Número de mostradores abiertos**: en general, a medida que más mostradores abiertos haya, más rápido avanzará la cola.

- **Tiempo que estaría en cola una persona que entrase ahora**: es el dato que se quiere predecir. Las unidades son de minutos.

En el apartado 5.6, se evaluará la relevancia de estos parámetros.

5.3.2.1. format_input

El script *format_input.py* recopilará información de los ficheros que contienen los datos de las entradas y las salidas a la cola, para generar una única *list*, llamada *queue*, en la que se juntará toda la información de entrada y salidas.

De esta manera, la *list queue* contendrá todas las entradas y salidas individualmente, ordenadas cronológicamente.

5.3.2.2. get_output

Esta función *get_output*, utilizará la *list queue* generada anteriormente, para calcular y almacenar todos los datos necesarios para la entrada al algoritmo de machine learning.

El funcionamiento es el siguiente: en primer lugar se inicializarán tres variables locales principales:

- **queueLength**: longitud en personas de la cola en cada momento. Inicializada a 0, ya que en el inicio la cola debe estar vacia.
- **lastExitTime**: tiempo de espera de la última persona en salir de la cola, en minutos. Se inicializa a infinito, ya que inicialmente nadie ha salido de la cola todavía.
- **output**: *list* en la que se almacenarán los datos.

A continuación, se iterará por todos los elementos de la *list queue*, en el orden cronológico en el que habían sido colocados anteriormente. Dependiendo de si es una entrada o una salida, se realizarán acciones diferentes:

- Si es una entrada: se añade a la *list output* un *dict* con la siguiente información:

    - **time**: objeto *datetime* con la fecha y hora en que se ha producido la entrada a la cola. Se obtiene directamente de la *list queue*.
    - **booths**: número de mostradores abiertos cuando se ha producido la entrada a la cola. Se obtiene directamente de la *list queue*.
    - **queueLength**: longitud de la cola, se obtiene del valor en esta iteración de la variable local *queueLength*.
    - **lastExitTime**: tiempo de espera de la última persona en salir de la cola, se obtiene del valor en esta iteración de la variable local *lastExitTime*.
    - **myExitTime**: tiempo de que ha tardado en salir esta persona, se inicializa a infinito, pero será modificado posteriormente y no se mantendrá en ningún caso como infinito.

    Finalmente, se incrementa el valor de la variable local *queue* en uno, ya que “ha entrado” una persona a la cola.

- Si es una salida: se calcula el tiempo que ha estado en cola la siguiente persona en salir, que se hace restando el valor del objeto *datetime* de la persona a la que le toca salir (que representa cuando entro esta persona a la cola), del tiempo que corresponde a esta salida de la *list queue*.

    A continuación, se actualiza la variable *lastExitTime* al tiempo de espera que ha tenido esta persona, el mismo que se acaba de calcular.

    Finalmente, se decrece el valor de *queueLength* en uno, ya que ha “salido” una persona de la cola.

Una vez acabado se tendrá la *list output* con todos los datos necesarios para la entrada al algoritmo de machine learning, solo faltará darles el formato correcto.

5.3.2.3. write_output

La función *write_output*, se encarga de darlo formato a los datos, para que pasen de estar en la *list output*, a un fichero *.csv* con la forma adecuada, descrita en el apartado 5.3.2.

Es importante destacar, que todos los *dicts* de la *list output*, que tengan como valor de *lastExitTime*: infinito, son de sujetos que entraron cuando aún no había salido nadie de la cola, y por tanto no tienen este valor correspondiente al tiempo de espera de la última persona en salir de la cola en cada momento. Por ello, se deben eliminar, ya que no contienen los datos completos necesarios para la entrada al algoritmo de machine learning.

5.4. Selección del algoritmo de machine learning

En este apartado, se seleccionará el algoritmo de machine learning a utilizar para predecir el tiempo de espera en cola en un aeropuerto. Para ello, primero se escogerá la librería a utilizar para implementar estos algoritmos y, a continuación, se compararán dos algoritmos para averiguar cuál da mejores resultados con los datos generados anteriormente en el apartado 5.3.2.

De los datos generados anteriormente, que corresponden a todo el año 2019, se utilizaran en este apartado los correspondientes a una semana, la primera semana de 2019. Esto es debido a que el tiempo de entrenamiento de los algoritmos de machine learning es lento y cuantos más datos, más tiempo se tarda en entrenar. Como en este apartado el objetivo es hacer una comparación entre algoritmos, no es necesario hacerlo con todos los datos de los que se dispone.

5.4.1. Selección de la librería a utilizar

Igual que en los apartados anteriores de este proyecto, el lenguaje de programación que se utilizará será Python.

Python dispone de varias librerías para implementar algoritmos de machine learning, en este caso se utilizará la librería scikit-learn. El motivo de escoger scikit-learn en lugar de otras opciones, como por ejemplo TensorFlow, es que permite utilizar algoritmos de machine learning de una forma más sencilla, ya que ya vienen implementados dentro de la librería [28]. Por ello, permitirá enfocarse más en la comparación entre algoritmos o la optimización de los parámetros del algoritmo.

5.4.2. Red neuronal

En este apartado se describirán las pruebas realizadas con redes neuronales. Este tipo de algoritmo ya ha sido explicado anteriormente en el apartado 1.4.

5.4.2.1. Descripción de las pruebas a realizar

Las pruebas a realizar, consistirán en utilizar redes neuronales, con diferentes configuraciones de capas y nodos para intentar ver, en una primera aproximación, que configuración da mejores resultados.

Las configuraciones de capas y nodos escogidas para esta prueba serán:

- Capas: 1, 2, 3, 4 y 5.
- Nodos por capa: 1, 5, 10, 14, 15, 20, 50 y 100.

Para implementar el algoritmo, se utilizará la función *MLPRegressor* que proporciona scikit-learn [29]. Esta función utiliza una red neuronal de múltiples capas de nodos, enfocada a la regresión, que nos permitirá ajustar diversos parámetros, entre los que se encuentra la configuración de capas y nodos. El resto de parámetros de la función se dejará por defecto, a excepción del número de iteraciones que se limitará a mil.

Para la evaluación del algoritmo se utilizará la validación cruzada, dividiendo los datos en diez grupos y, por tanto, entrenando y validando diez veces, siendo en cada prueba un grupo diferente el que valida y el resto entrenan. El funcionamiento de una validación cruzada está explicado en detalle en el apartado 1.6.1.

Para poder comparar las diferentes configuraciones, se medirán dos parámetros: el tiempo de ejecución y el error relativo medio. El tiempo de ejecución será el correspondiente a las diez pruebas de validación cruzada y el error relativo medio será el correspondiente a la media de error relativo de las diez pruebas de validación cruzada.

5.4.2.2. Resultados obtenidos

Los resultados obtenidos en estas pruebas, respecto al error relativo medio son los siguientes:

Tabla 5.1 Error relativo medio de las pruebas con redes neuronales

Nodos por capa \ Capas	1	2	3	4	5
1	0.1254	0.2464	0.2814	0.3002	0.2618
5	0.0886	0.0877	0.0859	0.0818	0.0813
10	0.0767	0.0637	0.0548	0.0487	0.0488
14	0.0667	0.0480	0.0397	0.0371	0.0349
15	0.0644	0.0464	0.0390	0.0349	0.0345
20	0.0576	0.0373	0.0314	0.0301	0.0303
50	0.0369	0.0264	0.0229	0.0222	0.0209
100	0.0289	0.0227	0.0199		

Como se observa en la **Tabla 5.1**, en general, a medida que se incrementa el número de capas y el número de nodos por capa, se reduce el error relativo, mejorando los resultados de la red neuronal.

El error relativo está calculado en función de tiempos en **minutos**, por lo que por ejemplo, el caso de 3 capas de 100 nodos, da un error relativo medio del 1,99% en minutos.

En lo que respecta al tiempo, los resultados obtenidos son los siguientes:

Tabla 5.2 Tiempo de ejecución en horas de las pruebas con redes neuronales

Nodos por capa \ Capas	1	2	3	4	5
1	0.06	0.06	0.07	0.08	0.08
5	0.13	0.35	0.82	0.68	1.09
10	0.42	0.52	0.98	1.49	2.17
14	0.53	0.99	1.14	1.87	2.49
15	0.68	0.90	1.16	1.78	2.55
20	0.74	0.93	1.40	2.27	4.03
50	1.23	1.28	3.22	5.13	7.92
100	1.85	5.65	16.18		

Como se puede observar en la **Tabla 5.2**, en general, a medida que se aumentan el número de capas y nodos por capa, se incrementa también el tiempo de ejecución.

Como conclusión de este apartado se puede decir que, en general, a medida que se consiguen mejores resultados (menor error relativo), se requiere más tiempo de ejecución para poder entrenar la red neuronal.

5.4.3. Máquina de vectores de soporte

En este apartado se describirán las pruebas realizadas con máquinas de vectores de soporte (SVM). El funcionamiento de las SVM, ha sido descrito en el apartado 1.3.2.

5.4.3.1. Descripción de las pruebas a realizar

Las pruebas a realizar en este apartado, consistirán en utilizar SVM con diferentes *kernels*, para comprobar si da buenos resultados con los datos de colas en aeropuertos.

Los diferentes *kernels* con los que se realizarán las pruebas son:

- Lineal.
- RBF (*Radial Basis Function*).
- Polinómico de grado 3.
- Polinómico de grado 5.

Para la implementación del algoritmo, se utilizará la función SVR de la librería scikit-learn [30]. Esta función utiliza una máquina de vectores de soporte como algoritmo de regresión. Los únicos parámetros de la función que se modificaran serán el *kernel* y el grado (este solo para los polinómicos), el resto de parámetros serán por defecto.

Tanto la manera de utilizar la *cross-validation*, como los parámetros a medir y la forma de medirlos, serán exactamente igual al apartado anterior de redes neuronales 5.4.2.

5.4.3.2. Resultados obtenidos

Los resultados obtenidos en estas pruebas, respecto al error relativo medio son los siguientes:

Tabla 5.3 Error relativo medio de las pruebas con máquinas de vectores de soporte

	Kernel			
	lineal	rbf	polinómico: grado 3	polinómico: grado 5
Error relativo	0.10587	0.04857	0.06520	0.04042

Como se puede observar en la **Tabla 5.3**, el mejor resultado se obtiene con el *kernel* polinómico de grado 5, seguido por el RBF.

En lo que respecta al tiempo, los resultados obtenidos son los siguientes:

Tabla 5.4 Tiempo de ejecución en horas de las pruebas con máquinas de vectores de soporte

	Kernel			
	lineal	rbf	polinómico: grado 3	polinómico: grado 5
Tiempo de ejecución	9.5310	18.7533	8.1607	14.3880

Se puede observar en la **Tabla 5.4**, que los tiempos de ejecución son en todos los casos muy elevados y, especialmente en el caso del *kernel* polinómico de grado 5 y el RBF.

5.4.4. Comparación y selección del algoritmo

Ahora que ya se tienen los resultados de los dos algoritmos, se puede realizar la comparación. Para facilitar esta comparación, se mostrarán todos los datos en la siguiente gráfica:

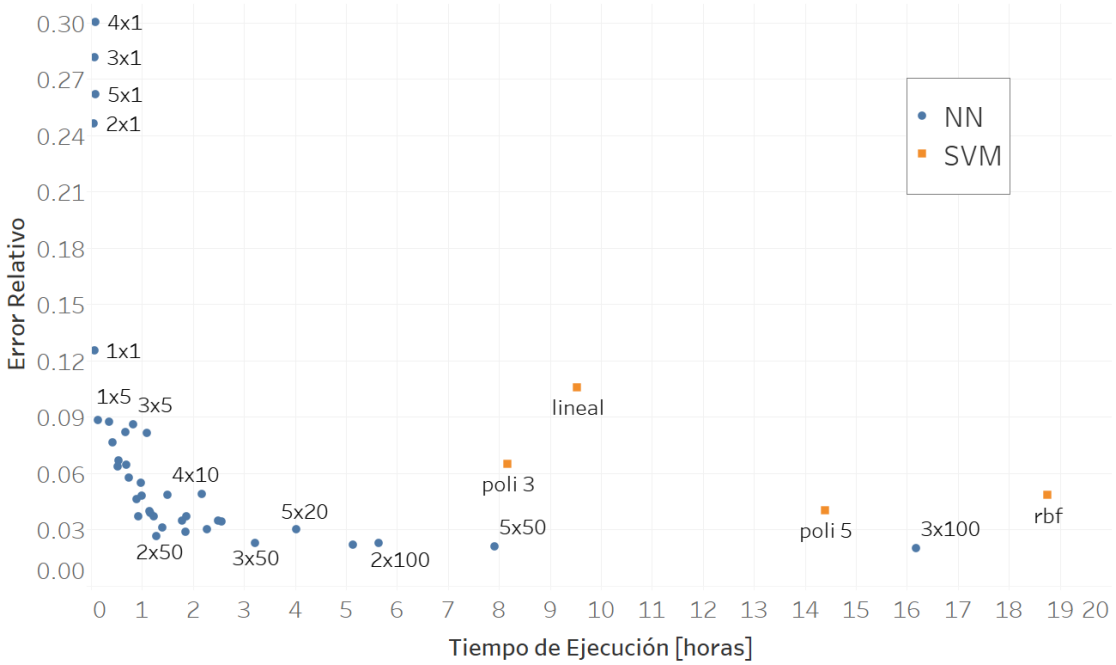

Fig. 5.3 Comparación redes neuronales y máquinas de vectores de soporte

En la **Fig. 5.3**, se puede ver como los resultados de las máquinas de vectores de soporte tienen un error relativo y un tiempo de ejecución superior que las redes neuronales. Por este motivo, se concluye que las redes neuronales son una mejor opción para este conjunto de datos.

Dentro de las redes neuronales, también hay diferencias dependiendo de la configuración y se debe escoger una opción con un buen balance entre error relativo y tiempo de ejecución, para utilizarla como referencia en pruebas posteriores. Por este motivo, se descartan los valores muy elevados tanto de tiempo de ejecución, como de error relativo y se obtiene la siguiente gráfica:

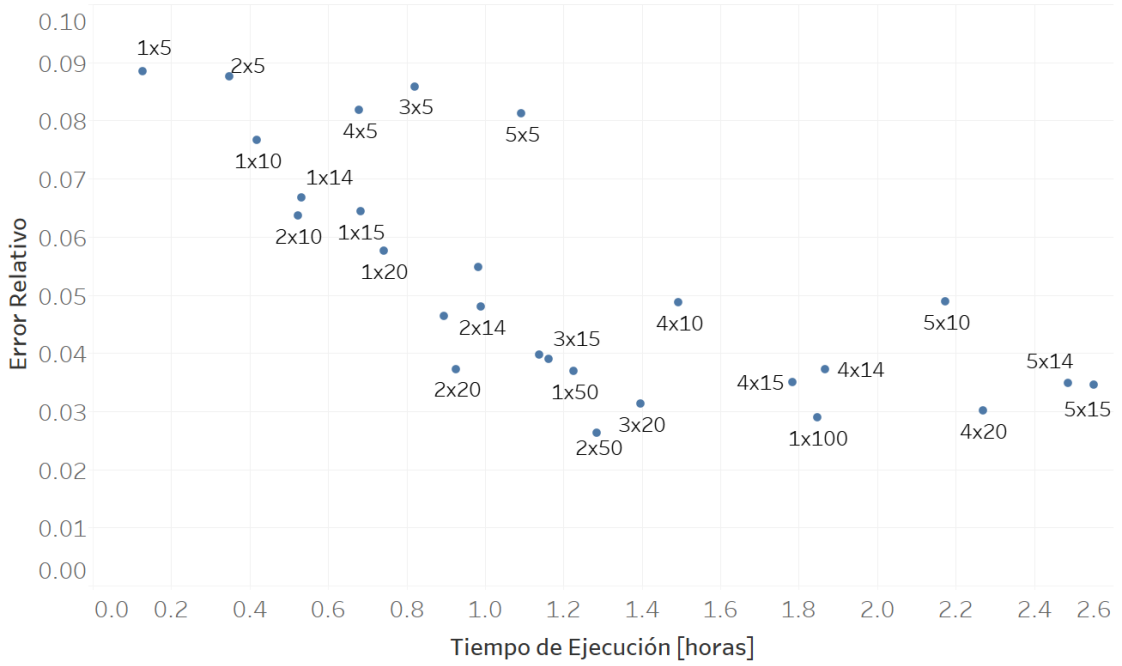

Fig. 5.4 Comparación mejores resultados red neuronal

Del conjunto de resultados mostrados en la **Fig. 5.4**, el mejor en cuanto a error relativo es el correspondiente a dos capas de cincuenta nodos cada una, este error relativo medio tiene un valor de **0,02635**. Esta prueba, también tiene un tiempo relativamente reducido de 1,2 horas. Por estos motivos, será la prueba escogida para optimizar en el siguiente apartado.

5.5. Optimización de la red neuronal

En este apartado se optimizará la red neuronal con la cual se obtuvieron mejores resultados en el apartado anterior, es decir, la red neuronal de dos capas de cincuenta nodos.

Para la optimización se modificarán diversos parámetros de la red neuronal, para ver si mejoran o no los resultados. Estos parámetros a modificar son: el número de nodos por capa, el ratio de aprendizaje, el parámetro alfa y la arquitectura de la red.

Finalmente, la configuración que de mejores resultados utilizando los datos correspondientes a una semana, se probará con los datos del año 2019 completo.

En este apartado se medirá el error medio relativo, así como también se utilizará una validación cruzada dividiendo en diez grupos para todas las pruebas. Todo esto se hará exactamente igual que como se ha descrito en el apartado 5.4.2.

5.5.1. Optimización de número de nodos, ratio de aprendizaje y alfa.

En este apartado se hará la primera prueba de optimización que consistirá en modificar el número de nodos, ratio de aprendizaje y parámetro alfa. Esto permitirá escoger la combinación de estos parámetros que proporcione los mejores resultados.

5.5.1.1. Definición del ratio de aprendizaje

El ratio de aprendizaje es el parámetro que ajusta el salto que dan los pesos (*W*) de la red neuronal en cada iteración [31]. Es un parámetro complicado de ajustar porque si se escoge un valor muy alto, puede que se salte por encima el valor mínimo de error para la red neuronal y no lo encuentre, pero un valor muy pequeño puede hacer que el entrenamiento tarde mucho en converger, e incluso que converja en un error mínimo local (**Fig 5.5**) en lugar de en el error mínimo real.

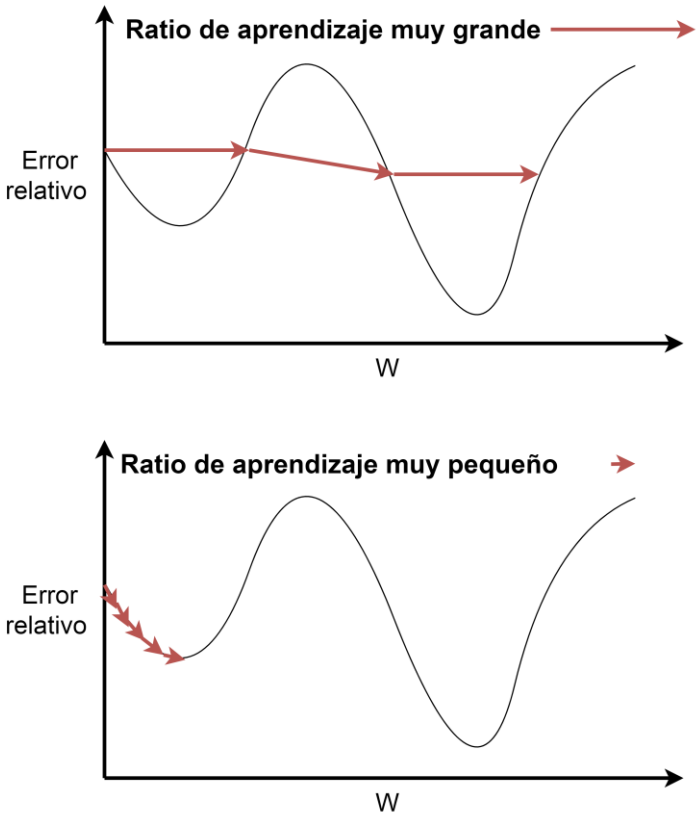

Fig. 5.5 Simplificación del ratio de aprendizaje

5.5.1.2. Definición del parámetro alfa

El parámetro alfa es un parámetro de regularización el cual modifica la función de coste que el algoritmo busca minimizar para penalizar la complejidad de la red neuronal y así evitar un sobreajuste (ver apartado 1.3.1). Valores muy altos pueden aumentar el error en las predicciones y valores muy bajos pueden dar lugar a sobreajustes.

5.5.1.3. Descripción de las pruebas a realizar

Las pruebas a realizar consistirán en modificar tres parámetros. El número de nodos por capa se variará alrededor de los cincuenta nodos por capa, para ver si mejoran el rendimiento encontrado en el mejor caso estudiado anteriormente. El ratio de aprendizaje y el parámetro alfa fueron anteriormente utilizados con sus parámetros por defecto, 0,001 y 0,0001 respectivamente y ahora se probarán otros valores para ver cuál su el efecto en el error relativo.

Los valores concretos que se probarán para cada parámetro son los siguientes:

- Nodos por capa (siempre dos capas): 40, 45, 50 y 55.
- Ratio de aprendizaje: 0,1, 0,05, 0,01, 0,005 y 0,001.
- Parámetro alfa: 0,005, 0,001, 0,0005 y 0,0001.

Para estos valores de los parámetros, se probarán todas las combinaciones posibles entre ellos. El motivo de esto, es que es posible que el mejor valor de cada uno de los parámetros si se analizan por separado, no sea el mismo que si se analizan todas las opciones posibles.

5.5.1.4. Resultados obtenidos

Para facilitar la visualización de los resultados de esta prueba, solo se mostrarán los mejores, es decir, los que tienen menor error relativo. Los resultados son los siguientes:

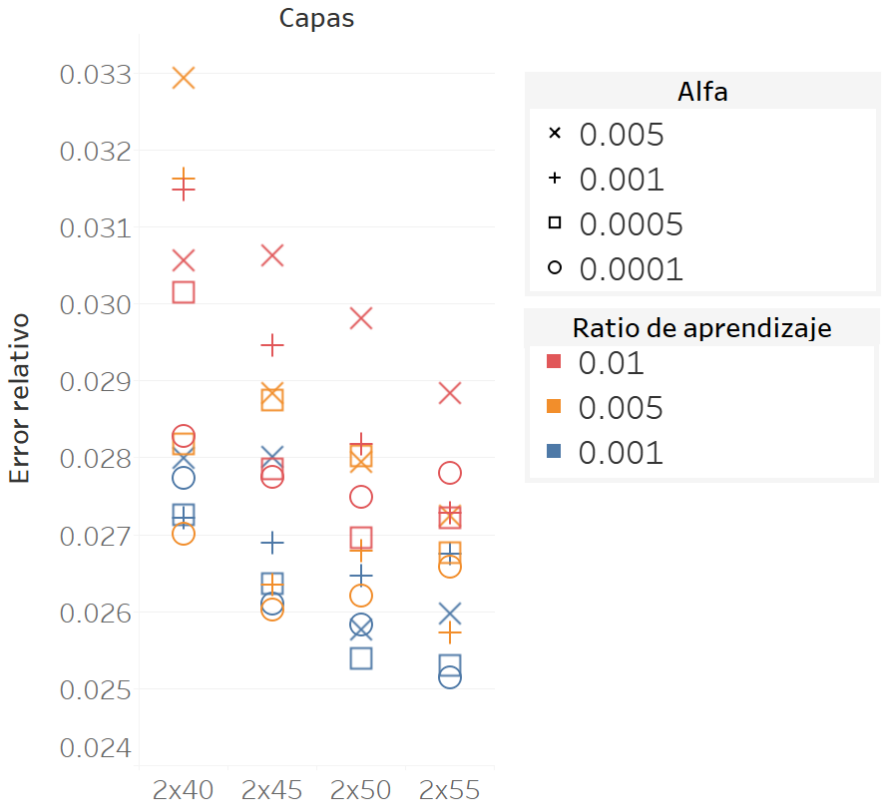

Fig. 5.6 Resultados optimización de nodos, ratio de aprendizaje y alfa (solo se muestran los mejores resultados obtenidos)

Como se puede ver en la **Fig 5.6**, los mejores resultados se obtienen, en general, cuando se disminuyen el ratio de aprendizaje y el parámetro alfa, y se aumenta el número de nodos por capa.

El mejor de los resultados obtenidos, corresponde a la prueba con dos capas de cincuenta y cinco nodos cada una, ratio de aprendizaje 0,001 y parámetro alfa 0,0001, con un resultado de error relativo de **0,02514**. Este resultado mejora el resultado obtenido anteriormente en el apartado 5.4.2.

Como conclusión, la arquitectura diseñada no tiende al *overfitting* y se puede prescindir de regularización.

5.5.1.5. Prueba con parámetro alfa igual a cero

Como se ha podido ver en el apartado anterior, al disminuir el ratio de aprendizaje y el parámetro alfa, en general, disminuye el error relativo.

Disminuir el ratio de aprendizaje hasta cero no tiene sentido, ya que no mejoraría los resultados de la red neuronal y se obtendría el mismo resultado

en todas las iteraciones (ver 5.5.1.1). Disminuir el parámetro alfa sí que puede tener sentido, ya que su principal función es reducir el sobreajuste, pero esto también se puede controlar con la validación cruzada. Por ello, se ha hecho una prueba con el parámetro alfa igual a cero.

Esta prueba se ha hecho con dos capas de cincuenta nodos cada una, ratio de aprendizaje 0,001 y parámetro alfa cero. El resultado obtenido ha sido un error relativo medio de **0,02494**, mejorando el resultado anteriormente obtenido en el apartado 5.5.1.4.

5.5.1.6. Efecto del ratio de aprendizaje en el tiempo

Disminuir el ratio de aprendizaje tiene como consecuencia un incremento en el tiempo de entrenamiento de la red neuronal. Para cuantificar este efecto se han realizado dos pruebas, que consistían en ver cuánto incrementa el tiempo de ejecución al incrementar el ratio de aprendizaje de 0,1 a 0,001, utilizando dos configuraciones de nodos, una de dos capas de cuarenta y otra de dos capas de cincuenta. Todas las pruebas se han realizado con un parámetro alfa de 0,0001.

Los resultados obtenidos han sido:

- Dos capas de cuarenta nodos: incremento del **58%** en el tiempo de ejecución.
- Dos capas de cincuenta nodos: incremento del **85%** en el tiempo de ejecución.

Como se puede ver el incremento de tiempo cuando se reduce el ratio de aprendizaje es considerable y es más grande a medida que se tienen más nodos en la red.

5.5.2. Optimización de la arquitectura

En este apartado se realizarán pruebas con diferentes arquitecturas de capas y nodos, para ver cuál es su efecto en el error relativo.

5.5.2.1. Descripción de las pruebas a realizar

Como en las pruebas anteriores se obtuvo el mejor resultado con dos capas de cincuenta y cinco nodos (ver apartado 5.5.1.4), ahora se realizarán pruebas para ver qué pasaría si se distribuyen esos mismos 110 nodos (2 capas de 55), en un número superior de capas.

Las arquitecturas seleccionadas para realizar las pruebas son las siguientes:

- Una capa: 110 nodos.

- Tres capas: dos de 37 nodos y una de 36.
- Cuatro capas: dos de 28 nodos y dos de 27.
- Cinco capas: todas de 22 nodos.

El resto de parámetros de la red neuronal serán: ratio de aprendizaje de 0,001 y parámetro alfa de 0,0001.

5.5.2.2. Resultados obtenidos

Los resultados obtenidos son los siguientes:

Tabla 5.5 Resultados con diferentes arquitecturas

Capas	Error relativo
(110)	0.02787
(55, 55)	0.02514
(37, 37, 36)	0.02537
(28, 28, 27, 27)	0.02666
(22, 22, 22, 22, 22)	0.02791

Como se observa en la **Tabla 5.5**, ninguno de los resultados supera los resultados de la prueba con dos capas de cincuenta y cinco nodos en las mismas condiciones (error relativo medio de 0,02514). Por lo tanto, no se ha obtenido ninguna mejora con estas arquitecturas.

5.5.3. Prueba de la red neuronal optimizada con datos de un año

En este apartado se realizará una prueba con la mejor configuración de parámetros obtenida después de la optimización de la red neuronal, pero con los datos completos del año 2019.

5.5.3.1. Descripción de la prueba a realizar

Después de haber optimizado la red neuronal, se puede concluir que, según las pruebas realizadas, la mejor configuración de parámetros de la red neuronal para los datos de colas en un aeropuerto correspondientes a la primera semana de 2019 es:

- Arquitectura: dos capas de cincuenta y cinco nodos cada una.

- Ratio de aprendizaje: 0,001.
- Parámetro alfa: 0.

Con estos parámetros, que son los mejores que se han podido obtener, se realizará la prueba con todos los datos del año 2019.

5.5.3.2. Resultado obtenido

El resultado obtenido, como media de las diez pruebas de validación cruzada, ha sido un error relativo de **0,07853**.

5.5.4. Conclusiones

En este apartado se ha podido ver cómo es la optimización de los diversos parámetros de una red neuronal.

El objetivo de esta optimización, era predecir con el menor error posible el tiempo de espera en cola en un aeropuerto. Finalmente se ha conseguido que con los datos de una semana, se tenga un error relativo medio de **0,02494** y con los datos de un año un error relativo de **0,07853**.

Si se tiene en cuenta que para todos los datos del año 2019, el tiempo de espera promedio es de 21,7 minutos, el error en la predicción supondría una variación de 1,7 minutos en la estimación del tiempo de espera, lo cual es un buen resultado.

5.6. Selección de parámetros de entrada

En este apartado se hará un análisis de los parámetros de entrada a la red neuronal, para analizar cuáles son los que tienen más importancia en lo que respecta a disminuir el error.

5.6.1. Descripción del método utilizado para la selección de parámetros de entrada

La selección de los parámetros de entrada, se realizará mediante la función *SelectKBest* de scikit-learn [33]. Esta función asigna a cada parámetro de entrada (también conocidos como *features*) una puntuación. El método que utiliza esta función para asignar la puntuación, es un análisis univariable, este método analiza cada variable de forma individual viendo cuál es su relación con el parámetro de salida. En función de que tan fuerte sea esta relación se asigna una puntuación mayor o menor.

Las pruebas a realizar, serán analizar todos los parámetros de entrada, con datos de una semana, un mes y un año.
Los resultados se normalizarán, para que resulten en un valor entre uno y cero, y así facilitar su comparación.

5.6.2. Resultados obtenidos

Los resultados obtenidos son los siguientes:

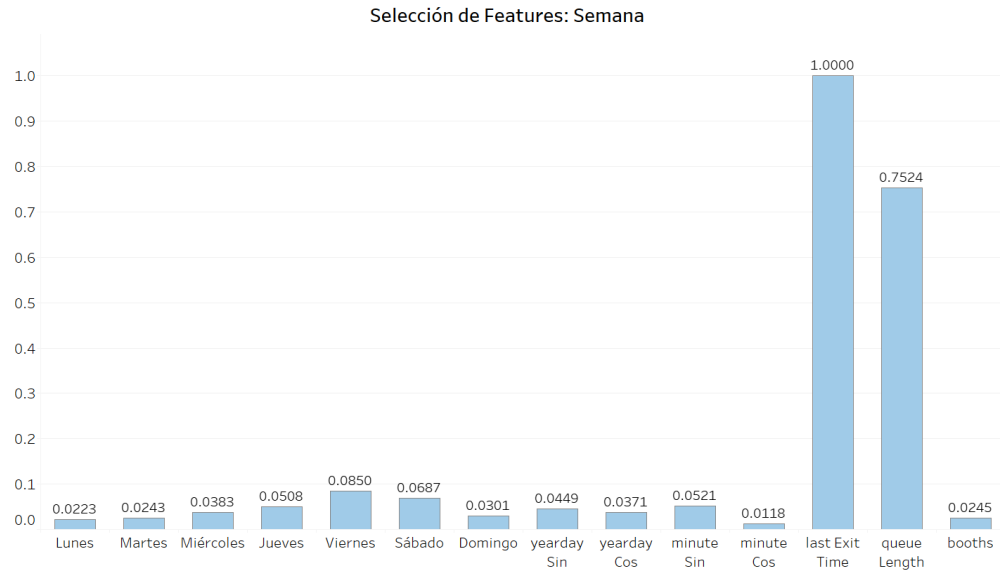

Fig. 5.7 Resultados selección de features: semana

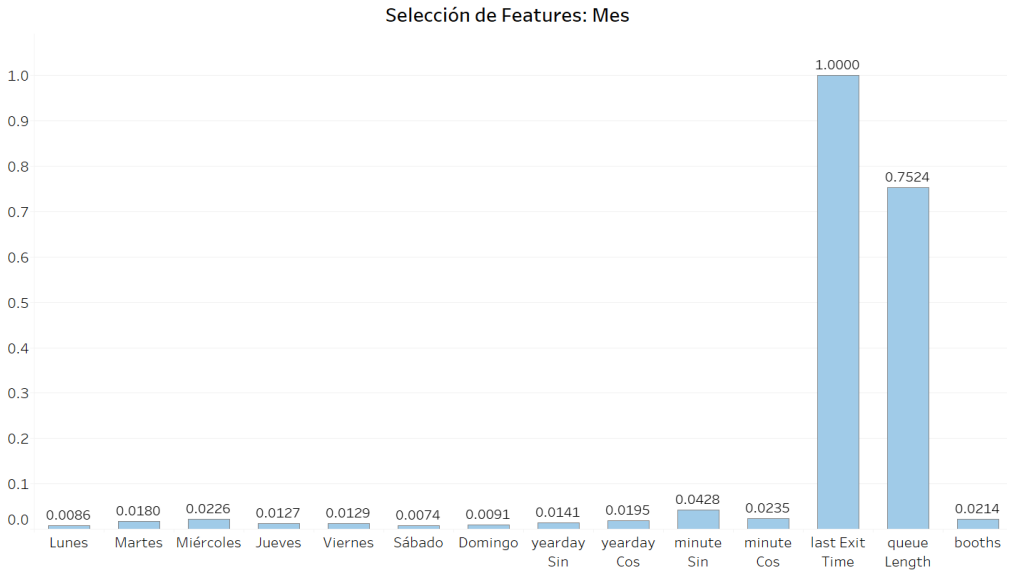

Fig. 5.8 Resultados selección de features: mes

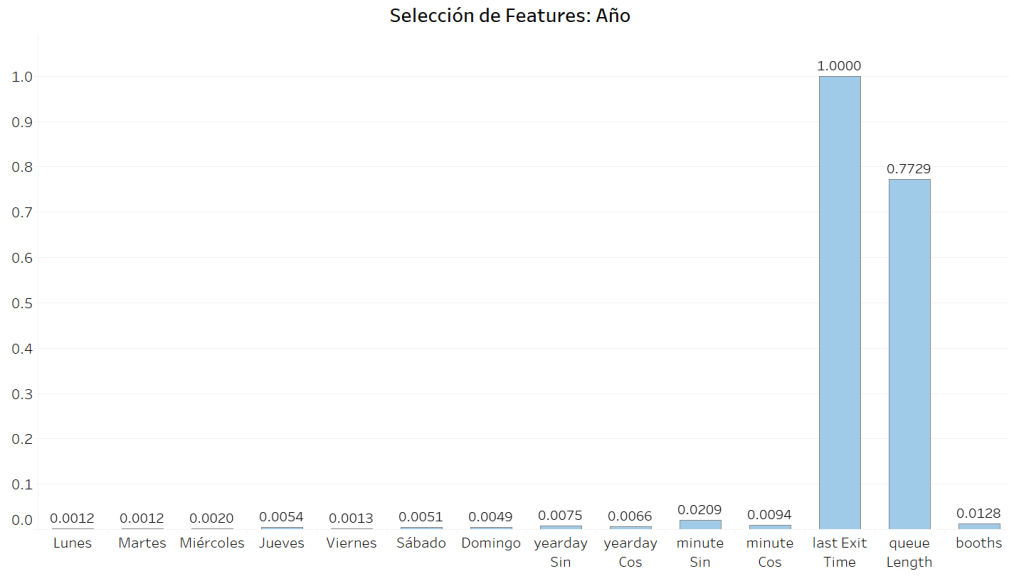

Fig. 5.9 Resultados selección de features: año

Como se puede observar en las figuras **Fig. 5.7**, **Fig. 5.8** y **Fig. 5.9,** los parámetros de entrada que tienen más importancia en todos los casos son el tiempo de espera de la última persona en salir de la cola y la longitud de la cola, teniendo el resto de parámetros una importancia muy inferior. También se puede ver cómo, a medida que aumentan los datos, ganan aún más importancia estos dos parámetros sobre los demás. Esto justifica el diseño del sistema que mide entradas y salidas, ya que esto aporta los parámetros más importantes.

Finalmente, para ver qué importancia tendría eliminar estos parámetros se hace una prueba con los datos de una semana, utilizando la red neuronal con la configuración que dio mejores resultados en el apartado 5.5. Estos son los resultados obtenidos respecto al error relativo:

- Mejor resultado anterior (sin eliminar parámetros): 0,02494
- Eliminando el tiempo de espera de la última persona en salir de la cola: 0,02571
- Eliminando el tiempo de espera de la última persona en salir de la cola y la longitud de la cola: 0,03027

Se observa que al eliminar estos parámetros los resultados de la red neuronal empeoran, pero aun así dan errores relativos bajos. Se puede decir que, si no se dispone del sistema de recuento de entradas y salidas, todavía seríamos capaces de hacer una buena predicción.

CAPÍTULO 6. CONCLUSIONES

En este apartado se explicarán las conclusiones que se extraen de este trabajo, así como las posibilidades de desarrollo que podrían existir en un futuro para continuar con lo que se ha hecho en este proyecto.

Los objetivos de este trabajo eran principalmente aumentar el conocimiento sobre machine learning, estudiar las posibles aplicaciones de machine en la gestión de un aeropuerto, implementar dos modelos de reconocimiento facial, uno *edge* y uno *cloud*, y posteriormente compararlos entre ellos; y también, desarrollar una herramienta que permitiera predecir el tiempo de espera en cola en un aeropuerto.

Sobre aumentar el conocimiento en machine learning, se ha conseguido tanto mediante la adquisición de conocimientos teóricos, como en el desarrollo práctico de diversos modelos, ya sean creados utilizando librerías (como en el CAPÍTULO 5), o implementados utilizando APIs (como Azure en el CAPÍTULO 3).

Las posibles aplicaciones a desarrollar utilizando machine learning para la gestión de un aeropuerto, se han centrado en la implementación del sistema de predicción de tiempo de espera en colas, que permitiría prepararse frente posibles incrementos en los tiempos de espera.

Por lo que respecta a los modelos de reconocimiento facial, en el caso edge, basado en la librería OpenCV de Python, se observó se debía tener cuidado con la calidad de las fotografías, especialmente con la iluminación, pero si era buena, se podían obtener buenos resultados. Para el modelo *cloud* ofrecido por Microsoft a través de Azure, los resultados obtenidos fueron excelentes acertando en todas las imágenes y, por ello, siendo superior al modelo *edge*.

En el análisis de tiempos y consumos de los modelos de reconocimiento facial, se mostró que tanto el tiempo como el consumo de energía eran notablemente inferiores cuando se utilizaba el modelo *cloud*, siendo el caso más favorable cuando se utilizaba con una conexión mediante cable.

Respecto a la herramienta para predecir el tiempo de espera en cola en un aeropuerto, tras una meticulosa selección de algoritmos y su optimización, se han podido obtener buenos resultados con un 7,9% de error relativo en minutos utilizando datos simulados de un año completo, generados a partir de una base de datos pública. Una predicción así podría dar pie a diversas aplicaciones, que podrían ser tanto informativas para los usuarios del aeropuerto, como de planificación en la gestión del aeropuerto.

Una de las posibles maneras de desarrollar en un futuro este trabajo, sería implementar diseño del modelo de predicción del tiempo de espera en colas utilizando Raspberry Pi en un entorno real, pero no ha sido posible por las excepcionales circunstancias en las que se ha desarrollado este último cuatrimestre del curso 2019/20.

Otro posible aspecto en el que se podría mejorar es utilizando la API de Azure que permite determinar si dos caras pertenecen o no a la misma persona, añadiéndola a los scripts tanto en la entrada como en la salida de la cola en las Raspberry Pi, para prevenir que se pueda contar dos veces la entrada o salida de una misma persona. Esto permitiría ganar un poco de precisión en los parámetros de entrada al modelo, pero también incrementaría la complejidad del sistema y se tendrían que tener en cuenta las posibles implicaciones legales que esto pueda tener.

Como trabajo futuro también queda abierta la puerta para estudiar otras aplicaciones de machine learning que sean útiles en la gestión de un aeropuerto.

BIBLIOGRAFÍA

[1] «Artificial intelligence to improve baggage handling in airports». [En línea]. Disponible en: https://www.airport-technology.com/news/artificial-intelligence-improve-baggage-handling-airports/. [Accedido: 05-jun-2020].

[2] «Frankfurt Airport is Using Machine Learning to Predict Aircraft Arrivals - Avionics». [En línea]. Disponible en: https://www.aviationtoday.com/2019/07/19/frankfurt-airport-using-machine-learning-predict-aircraft-arrivals/. [Accedido: 05-jun-2020].

[3] «Problemas en Barajas: colas, retrasos de vuelos y un excesivo e inusual control de equipajes». [En línea]. Disponible en: https://www.20minutos.es/noticia/3580370/0/colas-aeropuerto-barajas-t-4-control-acceso/. [Accedido: 05-jun-2020].

[4] «Why Insanely Long Airport Wait Times Are Likely the New Normal | The Takeaway | WNYC Studios». [En línea]. Disponible en: https://www.wnycstudios.org/podcasts/takeaway/segments/isanely-long-airport-wait-times-likely-new-normal. [Accedido: 05-jun-2020].

[5] «Aena prueba el sistema de reconocimiento facial en el proceso de embarque en el Aeropuerto de Menorca - Aena.es». [En línea]. Disponible en: http://www.aena.es/es/corporativa/aena-prueba-sistema-reconocimiento-facial-en-proceso---embarque-en-aeropuerto-menorca.html?p=1237548067436. [Accedido: 05-jun-2020].

[6] A. Geitgey, «Machine Learning is Fun! - Medium», 2014. [En línea]. Disponible en: https://medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471. [Accedido: 29-ago-2019].

[7] G. Seif, «The 5 Clustering Algorithms Data Scientists Need to Know - Medium», 2018. [En línea]. Disponible en: https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68. [Accedido: 30-ago-2019].

[8] «Train, Test, and Validation Sets explained - deeplizard», 2017. [En línea]. Disponible en: https://deeplizard.com/learn/video/Zi-0rlM4RDs. [Accedido: 30-ago-2019].

[9] A. Rosebrock, «Object detection with deep learning and OpenCV - PyImageSearch», 2017. [En línea]. Disponible en: https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/. [Accedido: 29-ago-2019].

[10] S. Ren, K. He, R. Girshick, y J. Sun, «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks», jun. 2015.

[11] J. Redmon, S. Divvala, R. Girshick, y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection», jun. 2015.

[12] W. Liu *et al.*, «SSD: Single Shot MultiBox Detector», dic. 2015.

[13] «BeginnersGuide/Download - Python Wiki». [En línea]. Disponible en: https://wiki.python.org/moin/BeginnersGuide/Download. [Accedido: 29-ago-2019].

[14] «Installation — PyPA». [En línea]. Disponible en: https://pip.pypa.io/en/stable/installing/. [Accedido: 11-sep-2019].

[15] «Installation NumPy — SciPy.org». [En línea]. Disponible en: https://scipy.org/install.html. [Accedido: 11-sep-2019].

[16] A. Rosebrock, «OpenCV Tutorials, Resources, and Guides -

PyImageSearch». [En línea]. Disponible en: https://www.pyimagesearch.com/opencv-tutorials-resources-guides/. [Accedido: 29-ago-2019].

[17] W. Price, «Install OpenCV 4.1.0 for Raspberry Pi 3 or 4 (Raspbian Buster) - GitHub», 2019. [En línea]. Disponible en: https://gist.github.com/willprice/abe456f5f74aa95d7e0bb81d5a710b60. [Accedido: 29-ago-2019].

[18] A. Rosebrock, «My imutils package: A series of OpenCV convenience functions - PyImageSearch», 2015. [En línea]. Disponible en: https://www.pyimagesearch.com/2015/02/02/just-open-sourced-personal-imutils-package-series-opencv-convenience-functions/. [Accedido: 11-sep-2019].

[19] C. E. Thomaz, «FEI Face Database - Centro Universitário FEI», 2006. .

[20] «¿Qué es Face API? - Azure Cognitive Services | Microsoft Docs». [En línea]. Disponible en: https://docs.microsoft.com/es-es/azure/cognitive-services/face/overview. [Accedido: 17-nov-2019].

[21] «Inicio rápido: Biblioteca cliente de Face para Python | Microsoft Docs». [En línea]. Disponible en: https://docs.microsoft.com/es-es/azure/cognitive-services/face/quickstarts/python-sdk. [Accedido: 17-nov-2019].

[22] «What is a Raspberry Pi?» [En línea]. Disponible en: https://www.raspberrypi.org/help/what- is-a-raspberry-pi/#:~:targetText=The Raspberry Pi is a,languages like Scratch and Python. [Accedido: 17-nov-2019].

[23] «FrontPage - Raspbian». [En línea]. Disponible en: https://www.raspbian.org/. [Accedido: 17-nov-2019].

[24] «Download Raspbian for Raspberry Pi». [En línea]. Disponible en: https://www.raspberrypi.org/downloads/raspbian/. [Accedido: 17-nov-2019].

[25] «Edge/Fog Computing: del Cloud hacia la computación en los dispositivos - Gradiant». [En línea]. Disponible en: https://www.gradiant.org/blog/edge-fog-computing-cloud/. [Accedido: 17-nov-2019].

[26] «time — Time access and conversions — Python 3.8.0 documentation». [En línea]. Disponible en: https://docs.python.org/3/library/time.html. [Accedido: 24-nov-2019].

[27] «Airport Wait Times». [En línea]. Disponible en: https://awt.cbp.gov/. [Accedido: 30-mar-2020].

[28] S. Raschka, «What is the main difference between TensorFlow and scikit-learn?» [En línea]. Disponible en: https://sebastianraschka.com/faq/docs/tensorflow-vs-scikitlearn.html. [Accedido: 14-may-2020].

[29] «sklearn.neural_network.MLPRegressor — scikit-learn 0.23.0 documentation». [En línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html. [Accedido: 14-may-2020].

[30] «sklearn.svm.SVR — scikit-learn 0.23.0 documentation». [En línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR. [Accedido: 14-may-2020].

[31] H. Zulkifli, «Understanding Learning Rates and How It Improves Performance in Deep Learning». [En línea]. Disponible en: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10. [Accedido: 18-may-2020].

[32] P. Gupta, «Regularization in Machine Learning - Towards Data Science». [En línea]. Disponible en: https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a. [Accedido: 18-may-2020].

[33] «sklearn.feature_selection.SelectKBest — scikit-learn 0.23.1 documentation». [En línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest. [Accedido: 20-may-2020].

[34] A. Rosebrock, «OpenCV Face Recognition - PyImageSearch», 2018. [En línea]. Disponible en: https://www.pyimagesearch.com/2018/09/24/opencv-face-recognition/. [Accedido: 29-ago-2019].

ANEXO A. CÓDIGOS Y RESULTADOS DEL RECONOCIMIENTO FACIAL

En este anexo se mostrarán en detalle los códigos utilizados para el reconocimiento facial, así como los resultados específicos de cada una de las pruebas realizadas.

A.1. Códigos para realizar el reconocimiento facial

Los códigos mostrados a continuación han sido extraídos de [34] y modificados parcialmente.

A.1.1. *extract_embeddings.py*

```
# importar los paquetes necesarios
from imutils import paths
import numpy as np
import argparse
import imutils
import pickle
import cv2
import os

# construir el analizador sintáctico de parámetros y
# analizar los parámetros
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--dataset", required=True,
    help="path to input directory of faces + images")
ap.add_argument("-e", "--embeddings", required=True,
    help="path to output serialized db of facial embeddings")
ap.add_argument("-d", "--detector", required=True,
    help="path to OpenCV's deep learning face detector")
ap.add_argument("-m", "--embedding-model", required=True,
    help="path to OpenCV's deep learning face embedding model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# cargar el detector de caras
print("[INFO] loading face detector...")
protoPath = os.path.sep.join([args["detector"], "deploy.prototxt"])
modelPath = os.path.sep.join([args["detector"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
detector = cv2.dnn.readNetFromCaffe(protoPath, modelPath)

# cargar el modelo para obtener los embeddings
print("[INFO] loading face recognizer...")
embedder = cv2.dnn.readNetFromTorch(args["embedding_model"])

# cargar la ruta de las imagenes de entrada de la base de datos
print("[INFO] quantifying faces...")
```

```
imagePaths = list(paths.list_images(args["dataset"]))

# inicializar la lista de embeddings extraídos y
# los nombres de las personas correspondientes
knownEmbeddings = []
knownNames = []

# inicializar el número total de caras procesadas
total = 0

# iterar sobre las rutas de las imágenes
for (i, imagePath) in enumerate(imagePaths):
    # extraer el nombre de la persona de la ruta de la imagen
    print("[INFO] processing image {}/{}".format(i + 1,
        len(imagePaths)))
    name = imagePath.split(os.path.sep)[-2]

    # cargar la imagen, redimensionarla para que tenga un ancho
    # de 600 píxeles (manteniendo la relación dimensional), y
    # luego guardar las dimensiones de la imagen
    image = cv2.imread(imagePath)
    image = imutils.resize(image, width=600)
    (h, w) = image.shape[:2]

    # construir un BLOB (objeto binario grande) a partir
    # de la imagen
    imageBlob = cv2.dnn.blobFromImage(
        cv2.resize(image, (300, 300)), 1.0, (300, 300),
        (104.0, 177.0, 123.0), swapRB=False, crop=False)

    # aplicar el detector facial de deep learning de OpenCV para
    # localizar caras en la imagen de entrada
    detector.setInput(imageBlob)
    detections = detector.forward()

    # asegurar que al menos se encontró una cara
    if len(detections) > 0:
        # asumiendo que cada imagen tiene una sola cara, encerrar
        # en un cuadrado el lugar donde es más probable que esté
        # situada
        i = np.argmax(detections[0, 0, :, 2])
        confidence = detections[0, 0, i, 2]

        # asegurar que la detección con mayor probabilidad esta por
        # encima del límite de probabilidad necesario (para evitar
        # que haya una detección falsa)
        if confidence > args["confidence"]:
            # computar las coordenadas (x,y) del cuadrado que
            # encerrará la cara
            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
            (startX, startY, endX, endY) = box.astype("int")

            # extraer la Región de Interés (ROI) de la cara y
            # sus dimensiones
            face = image[startY:endY, startX:endX]
            (fH, fW) = face.shape[:2]

            # asegurar que el ancho y largo de la cara sea
            # suficientemente grande
```

```
            if fW < 20 or fH < 20:
                continue

            # construir un BLOB para la ROI de la cara, después
            # pasar el BLOB a través del modelo que extrae los
            # embeddings y así obtener los 128 parámetros que
            # cuantifican la cara
            faceBlob = cv2.dnn.blobFromImage(face, 1.0 / 255,
                (96, 96), (0, 0, 0), swapRB=True, crop=False)
            embedder.setInput(faceBlob)
            vec = embedder.forward()

            # añadir el nombre de la persona y los
            # correspondientes embeddings de la cara a sus
            # respectivas listas
            knownNames.append(name)
            knownEmbeddings.append(vec.flatten())
            total += 1

# guardar los embeddings con sus respectivos nombres
print("[INFO] serializing {} encodings...".format(total))
data = {"embeddings": knownEmbeddings, "names": knownNames}
f = open(args["embeddings"], "wb")
f.write(pickle.dumps(data))
f.close()
```

Código A.1 *extract_embeddings.py*

Para ejecutar este código se debe utilizar el siguiente comando:

```
1.    python extract_embeddings.py --dataset dataset \
2.        --embeddings output/embeddings.pickle \
3.        --detector face_detection_model \
4.        --embedding-model openface_nn4.small2.v1.t7
```

Código A.2 ejecutar *extract_embeddings.py*

A.1.2. *train_model.py*

```
# importar los paquetes necesarios
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import SVC
import argparse
import pickle

# construir el analizador sintáctico de parámetros y
# analizar los parámetros
ap = argparse.ArgumentParser()
ap.add_argument("-e", "--embeddings", required=True,
```

```
    help="path to serialized db of facial embeddings")
ap.add_argument("-r", "--recognizer", required=True,
    help="path to output model trained to recognize faces")
ap.add_argument("-l", "--le", required=True,
    help="path to output label encoder")
args = vars(ap.parse_args())

# cargar los embeddings de las caras
print("[INFO] loading face embeddings...")
data = pickle.loads(open(args["embeddings"], "rb").read())

# codificar las etiquetas
print("[INFO] encoding labels...")
le = LabelEncoder()
labels = le.fit_transform(data["names"])

# entrenar el modelo que recibe los embeddings de las caras
# y entonces proporciona el reconocimiento facial
print("[INFO] training model...")
recognizer = SVC(C=1.0, kernel="linear", probability=True)
recognizer.fit(data["embeddings"], labels)

# guardar el modelo de reconocimiento facial
f = open(args["recognizer"], "wb")
f.write(pickle.dumps(recognizer))
f.close()

# guardar el codificador de etiquetas
f = open(args["le"], "wb")
f.write(pickle.dumps(le))
f.close()
```

Código A.3 *train_model.py*

Para ejecutar este código se debe utilizar el siguiente comando:

```
python train_model.py --embeddings output/embeddings.pickle \
    --recognizer output/recognizer.pickle \
    --le output/le.pickle
```

Código A.4 ejecutar *train_model.py*

A.1.3. recognize.py

```
# importar los paquetes necesarios
import numpy as np
import argparse
import imutils
import pickle
import cv2
import os
```

```
# construir el analizador sintáctico de parámetros y
# analizar los parámetros
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-d", "--detector", required=True,
    help="path to OpenCV's deep learning face detector")
ap.add_argument("-m", "--embedding-model", required=True,
    help="path to OpenCV's deep learning face embedding model")
ap.add_argument("-r", "--recognizer", required=True,
    help="path to model trained to recognize faces")
ap.add_argument("-l", "--le", required=True,
    help="path to label encoder")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# cargar el detector facial
print("[INFO] loading face detector...")
protoPath = os.path.sep.join([args["detector"], "deploy.prototxt"])
modelPath = os.path.sep.join([args["detector"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
detector = cv2.dnn.readNetFromCaffe(protoPath, modelPath)

# cargar el modelo para obtener los embeddings
print("[INFO] loading face recognizer...")
embedder = cv2.dnn.readNetFromTorch(args["embedding_model"])

# cargar el modelo de reconocimiento facial junto
# con el codificador de etiquetas
recognizer = pickle.loads(open(args["recognizer"], "rb").read())
le = pickle.loads(open(args["le"], "rb").read())

# cargar la imagen, redimensionarla para que tenga un ancho
# de 600 píxeles (manteniendo la relación dimensional), y
# luego guardar las dimensiones de la imagen
image = cv2.imread(args["image"])
image = imutils.resize(image, width=600)
(h, w) = image.shape[:2]

# construir un BLOB (objeto binario grande) a partir
# de la imagen
imageBlob = cv2.dnn.blobFromImage(
    cv2.resize(image, (300, 300)), 1.0, (300, 300),
    (104.0, 177.0, 123.0), swapRB=False, crop=False)

# aplicar el detector facial de deep learning de OpenCV para
# localizar caras en la imagen de entrada
detector.setInput(imageBlob)
detections = detector.forward()

# iterar sobre las caras detectadas
for i in range(0, detections.shape[2]):
    # extraer la probabilidad asociada a las detecciones
    confidence = detections[0, 0, i, 2]

    # eliminar detecciones débiles
    if confidence > args["confidence"]:
```

```
        # computar las coordenadas (x,y) del cuadrado que
        # encerrará la cara
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")

        # extraer la Región de Interés (ROI) de la cara y
        # sus dimensiones
        face = image[startY:endY, startX:endX]
        (fH, fW) = face.shape[:2]

        # asegurar que el ancho y largo de la cara sean
        # suficientemente grandes
        if fW < 20 or fH < 20:
            continue

        # construir un BLOB para la ROI de la cara, después
        # pasar el BLOB a través del modelo que extrae los
        # embeddings y así obtener los 128 parámetros que
        # cuantifican la cara
        faceBlob = cv2.dnn.blobFromImage(face, 1.0 / 255, (96, 96),
            (0, 0, 0), swapRB=True, crop=False)
        embedder.setInput(faceBlob)
        vec = embedder.forward()

        # realizar una clasificación para reconocer la cara
        preds = recognizer.predict_proba(vec)[0]
        j = np.argmax(preds)
        proba = preds[j]
        name = le.classes_[j]

        # dibujar el cuadrado que encerrará la cara, junto con
        # la probabilidad asociada
        text = "{}: {:.2f}%".format(name, proba * 100)
        y = startY - 10 if startY - 10 > 10 else startY + 10
        cv2.rectangle(image, (startX, startY), (endX, endY),
            (0, 0, 255), 2)
        cv2.putText(image, text, (startX, y),
            cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)

# enseñar la imagen de salida
cv2.imshow("Image", image)
cv2.waitKey(0)
```

Código A.5 *recognize.py*

Para ejecutar este código se debe utilizar el siguiente comando:

```
python recognize.py --detector face_detection_model \
    --embedding-model openface_nn4.small2.v1.t7 \
    --recognizer output/recognizer.pickle \
    --le output/le.pickle \
    --image images/victor1.jpg
```

Código A.6 ejecutar *recognize.py*

Dependiendo de la imagen que se quiera probar, se debe cambiar la última línea para seleccionar la imagen adecuada.

El resultado debería ser una imagen similar a esta:

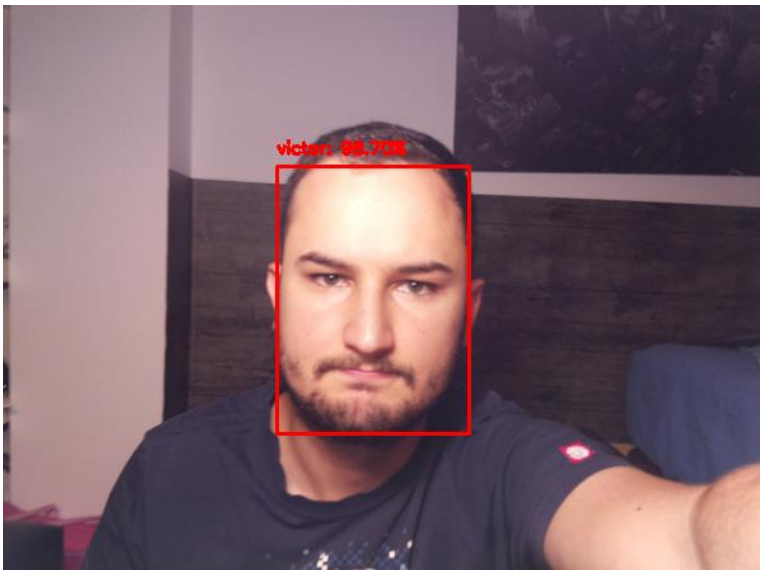

Fig. A.1 Ejemplo de imagen de salida del modelo

A.1.4. *recognize_video.py*

```
# importar los paquetes necesarios
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import pickle
import time
import cv2
import os

# construir el analizador sintáctico de parámetros y
# analizar los parámetros
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--detector", required=True,
    help="path to OpenCV's deep learning face detector")
ap.add_argument("-m", "--embedding-model", required=True,
    help="path to OpenCV's deep learning face embedding model")
ap.add_argument("-r", "--recognizer", required=True,
    help="path to model trained to recognize faces")
ap.add_argument("-l", "--le", required=True,
    help="path to label encoder")
```

```
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# cargar el detector facial
print("[INFO] loading face detector...")
protoPath = os.path.sep.join([args["detector"], "deploy.prototxt"])
modelPath = os.path.sep.join([args["detector"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
detector = cv2.dnn.readNetFromCaffe(protoPath, modelPath)

# cargar el modelo para obtener los embeddings
print("[INFO] loading face recognizer...")
embedder = cv2.dnn.readNetFromTorch(args["embedding_model"])

# cargar el modelo de reconocimiento facial junto
# con el codificador de etiquetas
recognizer = pickle.loads(open(args["recognizer"], "rb").read())
le = pickle.loads(open(args["le"], "rb").read())

# inicializar el vídeo, luego dejar que el sensor de la
# cámara se caliente
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)

# iniciar el contador de Fotogramas por Segundo (FPS)
fps = FPS().start()

# iterar sobre los fotogramas del vídeo
while True:
    # seleccionar un fotograma del vídeo
    frame = vs.read()

    # redimensionar el fotograma para que tenga un ancho
    # de 600 píxeles (manteniendo la relación dimensional), y
    # luego guardar las dimensiones de la imagen
    frame = imutils.resize(frame, width=600)
    (h, w) = frame.shape[:2]

    # construir un BLOB (objeto binario grande) a partir
    # de la imagen
    imageBlob = cv2.dnn.blobFromImage(
        cv2.resize(frame, (300, 300)), 1.0, (300, 300),
        (104.0, 177.0, 123.0), swapRB=False, crop=False)

    # aplicar el detector facial de deep learning de OpenCV para
    # localizar caras el fotograma de entrada
    detector.setInput(imageBlob)
    detections = detector.forward()

    # iterar sobre las caras detectadas
    for i in range(0, detections.shape[2]):
        # extraer la probabilidad asociada a las detecciones
        confidence = detections[0, 0, i, 2]

        # eliminar detecciones débiles
        if confidence > args["confidence"]:
            # computar las coordenadas (x,y) del cuadrado que
```

```
            # encerrará la cara
            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
            (startX, startY, endX, endY) = box.astype("int")

            # extraer la Región de Interés (ROI) de la cara y
            # sus dimensiones
            face = frame[startY:endY, startX:endX]
            (fH, fW) = face.shape[:2]

            # asegurar que el ancho y largo de la cara sean
            # suficientemente grandes
            if fW < 20 or fH < 20:
                continue

            # construir un BLOB para la ROI de la cara, después
            # pasar el BLOB a través del modelo que extrae los
            # embeddings y así obtener los 128 parámetros que
            # cuantifican la cara
            faceBlob = cv2.dnn.blobFromImage(face, 1.0 / 255,
                (96, 96), (0, 0, 0), swapRB=True, crop=False)
            embedder.setInput(faceBlob)
            vec = embedder.forward()

            # realizar una clasificación para reconocer la cara
            preds = recognizer.predict_proba(vec)[0]
            j = np.argmax(preds)
            proba = preds[j]
            name = le.classes_[j]

            # dibujar el cuadrado que encerrará la cara, junto con
            # la probabilidad asociada
            text = "{}: {:.2f}%".format(name, proba * 100)
            y = startY - 10 if startY - 10 > 10 else startY + 10
            cv2.rectangle(frame, (startX, startY), (endX, endY),
                (0, 0, 255), 2)
            cv2.putText(frame, text, (startX, y),
                cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)

    # actualizar el contador de FPS
    fps.update()
    # enseñar el fotograma de salida
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF
    # cuando la tecla 'q' sea pulsada, salir del bucle
    if key == ord("q"):
        break

# parar el contador de FPS y mostrar la información
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# cerrar todo
cv2.destroyAllWindows()
vs.stop()
```

Código A.7 recognize_video.py

Para ejecutar este código se debe utilizar el siguiente comando:

```
python recognize_video.py --detector face_detection_model \
    --embedding-model openface_nn4.small2.v1.t7 \
    --recognizer output/recognizer.pickle \
    --le output/le.pickle
```

Código A.8 ejecutar recognize_video.py

A.2. Descripción detallada y resultados de las pruebas realizadas

En este apartado se describirán en detalle las pruebas realizadas y los resultados obtenidos, en los diferentes escenarios que se plantean.

A.2.1. 15 fotografías sujeto Víctor y 150 fotografías de desconocidos en diferentes posturas

Para esta prueba fueron utilizadas las 15 fotografías del sujeto *Víctor*, descritas en el apartado 2.3.1.1 y las 150 fotografías de desconocidos en diferentes posturas, descritas en el apartado 2.3.1.4.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 10 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en diferentes posturas.
- Pruebas: 5 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en diferentes posturas.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.1 Matriz de confusión 5 fotografías *Víctor*, 50 fotografías desconocidos en diferentes posturas

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	2,33	2,67
	Desconocido	0,33	49,67

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 94,6 %
- Precisión: 91,7 %
- *Recall*: 46,7 %

A.2.2. 15 fotografías sujeto Víctor y 150 fotografías de desconocidos en la misma postura

Para esta prueba fueron utilizadas las 15 fotografías del sujeto *Víctor*, descritas en el apartado 2.3.1.1 y las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 10 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en la misma postura.
- Pruebas: 5 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en la misma postura.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.2 Matriz de confusión 5 fotografías *Víctor*, 50 fotografías desconocidos en la misma postura

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	2	3
	Desconocido	1	49

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 92,7 %
- Precisión: 50,0 %
- *Recall*: 40,0 %

A.2.3. 15 fotografías sujeto Víctor con mejor iluminación y 150 fotografías de desconocidos en diferentes posturas

Para esta prueba fueron utilizadas 15 de las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en diferentes posturas, descritas en el apartado 2.3.1.4.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 10 fotografías sujeto *Víctor* con mejor iluminación y 100 fotografías de desconocidos en diferentes posturas.
- Pruebas: 5 fotografías sujeto *Víctor* con mejor iluminación y 50 fotografías de desconocidos en diferentes posturas.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.3 Matriz de confusión 5 fotografías *Víctor*, 50 fotografías desconocidos en diferentes posturas

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	5	0
	Desconocido	0,33	49,67

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 99,4 %
- Precisión: 94,4 %
- *Recall*: 100 %

A.2.4. 15 fotografías sujeto Víctor con mejor iluminación y 150 fotografías de desconocidos en la misma postura

Para esta prueba fueron utilizadas 15 de las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 10 fotografías sujeto *Víctor* con mejor iluminación y 100 fotografías de desconocidos en la misma postura.
- Pruebas: 5 fotografías sujeto *Víctor* con mejor iluminación y 50 fotografías de desconocidos en la misma postura.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.4 Matriz de confusión 5 fotografías *Víctor*, 50 fotografías desconocidos en la misma postura

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	4,67	0,33
	Desconocido	0,33	49,67

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 98,8 %
- Precisión: 93,3 %
- *Recall*: 93,3 %

A.2.5. 30 fotografías sujeto Víctor y 150 fotografías de desconocidos en diferentes posturas

Para esta prueba fueron utilizadas las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en diferentes posturas, descritas en el apartado 2.3.1.4.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 20 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en diferentes posturas.
- Pruebas: 10 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en diferentes posturas.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.5 Matriz de confusión 10 fotografías *Víctor*, 50 fotografías desconocidos en diferentes posturas

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	8,33	2,33
	Desconocido	0,33	49,67

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 95,6 %
- Precisión: 97,0 %
- *Recall*: 78,6 %

A.2.6. 30 fotografías sujeto Víctor con mejor iluminación y 150 fotografías de desconocidos en la misma postura

Para esta prueba fueron utilizadas las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 20 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en la misma postura.
- Pruebas: 10 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en la misma postura.

La matriz de confusión obtenida al realizar las pruebas fue la siguiente:

Tabla A.6 Matriz de confusión 10 fotografías *Víctor*, 50 fotografías desconocidos en la misma postura

		Predicción	
		Víctor	Desconocido
Entrada	Víctor	8,67	1,33
	Desconocido	0,67	49,33

Los resultados de los distintos parámetros para la evaluación fueron los siguientes:

- Exactitud: 96,7 %
- Precisión: 93,9 %
- *Recall*: 86,7 %

A.2.7. Variación del número de fotografías del sujeto a identificar

Para esta prueba fueron utilizadas las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 2/3 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en la misma postura.
- Pruebas: 1/3 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en la misma postura.

El número de fotografías del sujeto a identificar (*Víctor*) era inicialmente de 30 y se fue disminuyendo de 3 en 3 fotografías hasta llegar a 6.

Los resultados de los distintos parámetros para la evaluación en las diferentes pruebas, fueron los siguientes:

Tabla A.7 Valores obtenidos de los parámetros de evaluación en las diferentes pruebas de variación del número de fotografías del sujeto a identificar

Número fotografías del sujeto a identificar	Exactitud	Precisión	Recall
6	98,72 %	83,33 %	83,33 %
9	98,11 %	87,5 %	77,78 %
12	99,38 %	92,31 %	100 %
15	98,79 %	93,33 %	93,33 %
18	98,22 %	91,74 %	90 %
21	95,91 %	85 %	80,95 %
24	95,98 %	86,96 %	83,33 %
27	97,18 %	92,31 %	88,89 %
30	96,11 %	89,66 %	86,66 %

A.2.8. Variación del número de fotografías de desconocidos

Para esta prueba fueron utilizadas las 30 fotografías del sujeto *Víctor* con mejor iluminación, descritas en el apartado 2.3.1.2 y las 150 fotografías de desconocidos en la misma postura, descritas en el apartado 2.3.1.5.

El entrenamiento se realizó utilizando la validación cruzada, con la siguiente distribución en las fotografías:

- Entrenamiento: 2/3 fotografías sujeto *Víctor* y 100 fotografías de desconocidos en la misma postura.
- Pruebas: 1/3 fotografías sujeto *Víctor* y 50 fotografías de desconocidos en la misma postura.

El número de fotografías de desconocidos era inicialmente de 198 y se fueron disminuyendo hasta llegar a 33.

Los resultados de los distintos parámetros para la evaluación en las diferentes pruebas, fueron los siguientes:

Tabla A.8 Valores obtenidos de los parámetros de evaluación en las diferentes pruebas de variación del número de fotografías de desconocidos

Número fotografías desconocidos	Exactitud	Precisión	Recall
33	88,89 %	87,10 %	90 %
48	91,03 %	87,10 %	90 %
66	90,63 %	81,82 %	90 %
81	93,69 %	87,10 %	90 %
99	93,80 %	86,67 %	86,67 %
114	94,44 %	86,67 %	86,67 %
132	94,44 %	88,89 %	80 %
147	96,05 %	89,66 %	86,67 %
165	96,41 %	89,66 %	86,67 %
180	96,67 %	89,66 %	86,67 %
198	97,37 %	96,15 %	83,33 %

ANEXO B. CÓDIGOS PREDICCIÓN COLAS

En este anexo se mostrarán los códigos utilizados para todo el proceso de predicción del tiempo de espera de una cola en un aeropuerto

B.1. Códigos para el procesado de datos

A continuación, se mostraran los scripts de Python utilizados para procesar los datos de la cola. Estos scripts han sido explicados en detalle en el apartado 5.3.

B.1.1. formatoRPi.py

```
import datetime
import random
import numpy
import csv
import os

# Ficheros csv de entrada y salida
INPUT_CSV = "LAX-international-year.csv"
OUTPUT_ARRIVALS_CSV = "LAX-international-arrivals-RPi-YEAR.csv"
OUTPUT_DEPARTURES_CSV = "LAX-international-departures-RPi-YEAR.csv"

def hour_Poisson_arrivals(times, lambd, previous, avgWaitTime, booths):
   """ Genera lista de timestamps con llegadas exponenciales a partir de la fecha ini
cial.
   Proporciona la lista de llegadas para una franja de una hora

   Parameters
   ----------
   times: list
      Lista a la cual se añaden las llegadas
   lambd: float
      Tasa de llegadas (llegadas/segundo)
   previous: datetime
      Fecha inicial en forma de objeto datetime
   avgWaitTime: int
      Tiempo medio de espera para esa hora (minutos/salida)
   booths: int
      Número de mostradores abiertos en esa hora

   Returns
   -------
   Lista de objetos datetime representado las llegadas generadas.
   También se añaden los valores de avgWaitTime y booths si modificar,
   se utilizarán en otras funcionas
   """

   # Mientras el tiempo transcurrido sea menor que una hora
   t = random.expovariate(lambd)
   while t < 3600.0:

      # Siguiente llegafa en tiempo aleatorio (distr. exp)
      times.append(["arrival", previous + datetime.timedelta(seconds=t), avgWaitTime,
 booths])
```

```
        t += random.expovariate(lambd)

    return times

def set_arrivals(arrivals):
    """ Obtiene los datos y los procesa para generar una lista de llegadas en el forma
to deseado

    Parameters
    ----------
    arrivals: list
        Lista a la cual se añaden las llegadas
    INPUT_CSV: str
        archivo que contiene los datos sin procesar

    Returns
    -------
    Lista con las llegadas en formato:
        ["arrival", datetime, avgWaitTime, booths]
    """
    global INPUT_CSV

    # se abre el fichero csv con los datos
    with open(os.path.join(os.path.dirname(__file__), INPUT_CSV)) as csvfile:
        rawData = csv.reader(csvfile, delimiter=';', quotechar='|')

        # para todas las filas del fichero
        for row in rawData:
            # se seleccionan y dan formato a los datos relevantes
            date = row[2].split("/")
            year, month, day, hour = int(date[2]), int(date[0]), int(date[1]), int(row[3
][:2])
            time = datetime.datetime(year,month,day,hour,0,0)
            avgWaitTime = int(row[8])
            nArrivals = int(row[18]) - int(row[17])
            booths = int(row[20])

            # se generan individualmente las llegadas con los datos que se poseen
            lambd = float(nArrivals/3600.0)
            hour_Poisson_arrivals(arrivals, lambd, time, avgWaitTime, booths)

    return arrivals

def set_departures(arrivals, departures):
    """ Genera las salidas de la cola a partir de las entradas

    Parameters
    ----------
    arrivals: list
        Lista con las llegadas
    departures
        lista a la que se añaden las salidas

    Returns
    -------
    Lista con las salidas en formato:
        ["departure", datetime]
    Ordenadas por fecha
    """

    for a in arrivals:
        mu = 0

        if a[2] == 0:
            # Si a[2] (tiempo medio de espera) es igual a cero,
            # es porque se ha redondeado a minutos y es inferior a
```

```
            # 30 segundos.
            # Como pasa en pocos casos, se considerará igual a 30
            # segundos para poder generar la salida.
            mu = 1/30
         else:
            mu = 1/(a[2]*60)

         departures.append(["departure", a[1] + datetime.timedelta(seconds=random.expova
riate(mu))])

   departures.sort(key=lambda d: d[1])
   return departures

def write_csv_data(arrivals, departures):
   """ Escribe los datos obtenidos en el formato correcto en ficheros .csv

   Parameters
   ----------
   arrivals: list
      Lista con las llegadas
   departures
      Lista con las salidas

   """
   global OUTPUT_ARRIVALS_CSV
   global OUTPUT_DEPARTURES_CSV

   with open(os.path.join(os.path.dirname(__file__), OUTPUT_ARRIVALS_CSV), 'w', newli
ne='') as csvfile:
      writer = csv.writer(csvfile, delimiter=';')
      for a in arrivals:
         writer.writerow([str(a[0])] + a[1].strftime("%Y;%m;%d;%H;%M;%S.%f").split(';
') + [str(a[3])])

   with open(os.path.join(os.path.dirname(__file__), OUTPUT_DEPARTURES_CSV), 'w', new
line='') as csvfile:
      writer = csv.writer(csvfile, delimiter=';')
      for d in departures:
         writer.writerow([str(d[0])] + d[1].strftime("%Y;%m;%d;%H;%M;%S.%f").split(';
'))

def main():
   """
   Este script utiliza los datos de la web: https://awt.cbp.gov/
   Los convierte en un formato como el que tendrían los datos si se obtuvieran con RP
is

   Formato
   -------
   Llegadas:
      ["arrival"; año; mes; día; hora; minutos; segundos; número de mostradores]
   Salidas
      ["departure"; año; mes; día; hora; minutos; segundos]
   """
   arrivals = []
   departures = []

   set_arrivals(arrivals)
   set_departures(arrivals, departures)
   write_csv_data(arrivals, departures)

if __name__ == "__main__":
   main()
```

Código B.1 *extract_embeddings.py*

B.1.2. formatoML.py

```
import datetime
import random
import numpy
import math
import csv
import os

# Ficheros csv de entrada y salida
INPUT_ARRIVALS_CSV = "LAX-international-arrivals-RPi-YEAR.csv"
INPUT_DEPARTURES_CSV = "LAX-International-departures-RPi-YEAR.csv"
OUTPUT_CSV = "LAX-international-YEAR-toML.csv"

def format_input(queue):
    """
    Obtiene los datos de llegadas y salidas y los procesa para generar una
    lista de dicts de la cola, con entradas y salidas en el formato deseado

    Parameters
    ----------
    INPUT_ARRIVALS_CSV: str
        Nombre del fichero que contiene las llegadas
    INPUT_DEPARTURES_CSV: str
        Nombre del fichero que contiene las salidas

    Returns
    -------
    Lista de dicts ordenada cronológicamente con las entradas y salidas a la cola en
formato:
        arrivals
            {action': str, 'time': datetime, 'booths': int}
        departures
            {action': str, 'time': datetime}

    """
    global INPUT_ARRIVALS_CSV
    global INPUT_DEPARTURES_CSV

    # se abre el ficheros csv con los datos de llegadas
    with open(os.path.join(os.path.dirname(__file__), INPUT_ARRIVALS_CSV)) as csvArri
vals:
        rawArrivals = csv.reader(csvArrivals, delimiter=';', quotechar='|')
        # para todas las filas del fichero
        for arrivalRow in rawArrivals:
            # se añaden a la lista de la cola
            year, month, day = int(arrivalRow[1]), int(arrivalRow[2]), int(arrivalRow
[3])
            hour, minute = int(arrivalRow[4]), int(arrivalRow[5])
            second, microsecond = int(arrivalRow[6].split(".")[0]), int(arrivalRow[6]
.split(".")[1])

            time = datetime.datetime(year,month,day,hour,minute,second,microsecond)

            queue.append({'action': arrivalRow[0], 'time': time, 'booths': int(arriva
lRow[7])})

    # se abre el ficheros csv con los datos de salidas
    with open(os.path.join(os.path.dirname(__file__), INPUT_DEPARTURES_CSV)) as csvDe
partures:
        rawDepartures = csv.reader(csvDepartures, delimiter=';', quotechar='|')
        # para todas las filas del fichero
        for departureRow in rawDepartures:
            # se añaden a la lista de la cola
```

```
            year, month, day = int(departureRow[1]), int(departureRow[2]), int(depart
ureRow[3])
            hour, minute = int(departureRow[4]), int(departureRow[5])
            second, microsecond = int(departureRow[6].split(".")[0]), int(departureRo
w[6].split(".")[1])

            time = datetime.datetime(year,month,day,hour,minute,second,microsecond)

            queue.append({'action': departureRow[0], 'time': time})

    # se ordena la cola
    queue.sort(key=lambda d: d['time'])

    return queue

def get_output(queue):
    queueLength = 0
    lastExitTime = float("inf")
    output = []
    exitIndex = -1

    for q in queue:
        if q['action'] == 'arrival':
            output.append({'time': q['time'],
                           'booths': q['booths'],
                           'queueLength': queueLength,
                           'lastExitTime': lastExitTime,
                           'myExitTime': float("inf")})

            queueLength += 1

        elif q['action'] == 'departure':

            exitIndex += 1
            lastExitTimeTimeDelta = q['time'] - output[exitIndex]['time']
            lastExitTime = int(lastExitTimeTimeDelta.total_seconds()/60)
            output[exitIndex]['myExitTime'] = lastExitTime

            queueLength -= 1

    return output

def write_output(output):
    ''''' Selecciona, da formato y escribe en un .csv los datos obtenidos

    Paramenters
    -----------
    output: list of dicts
        Contiene los datos de la cola
    '''

    def format_weekday(time):
        """
        Convierte el día de la semana desde el objeto datetime de entrada
        a un one-hot vector

        Parameters
        ----------
        time: datetime
            datetime del cual se quiere extraer el día de la semana

        Returns
        -------
        one-hot vector
            "1;0;0;0;0;0;0"    Lunes
            "0;1;0;0;0;0;0"    Martes
```

```
        """
        weekday = time.weekday()
        switcher = {
            0: [1,0,0,0,0,0,0],    # Lunes
            1: [0,1,0,0,0,0,0],    # Martes
            2: [0,0,1,0,0,0,0],    # Miércoles
            3: [0,0,0,1,0,0,0],    # Jueves
            4: [0,0,0,0,1,0,0],    # Viernes
            5: [0,0,0,0,0,1,0],    # Sábado
            6: [0,0,0,0,0,0,1]     # Domingo
        }
        return switcher.get(weekday)

    def format_yearday(time):
        # se obtiene el número de día del año (1-366)
        yearday = time.timetuple().tm_yday

        # se transforma a un valor entre 0 y 2pi
        transformedYearday = 2*math.pi*(yearday-1)/365

        # se obtiene un valor para seno y coseno entre -1 y 1
        yeardaySin = math.sin(transformedYearday)
        yeardayCos = math.cos(transformedYearday)

        return str(yeardaySin), str(yeardayCos)

    def format_minute(time):
        # se obtiene el minuto del día (0-1440)
        minute = time.minute + time.hour * 60

        # se transforma a un valor entre 0 y 2pi
        transformedMinute = 2*math.pi*minute/1440

        # se obtiene un valor para seno y coseno entre -1 y 1
        minuteSin = math.sin(transformedMinute)
        minuteCos = math.cos(transformedMinute)

        return str(minuteSin), str(minuteCos)

    global OUTPUT_CSV

    # se abre el fichero para escribir los resultados
    with open(os.path.join(os.path.dirname(__file__), OUTPUT_CSV), 'w', newline='') a
s csvfile:
        writer = csv.writer(csvfile, delimiter=';')
        for i in output:
            # Si no ha salido nadie antes que esta persona entrase
            # a la cola i["lastExitTime"] será infinito, no estarán completos
            # estos datos, y por tanto, se descartan
            if i["lastExitTime"] != float('inf'):
                # se da formato a los datos
                weekday = format_weekday(i["time"])
                yeardaySin, yeardayCos = format_yearday(i["time"])
                minuteSin, minuteCos = format_minute(i["time"])
                lastExitTime = str(i["lastExitTime"])
                queueLength = str(i["queueLength"])
                booths = str(i["booths"])
                myExitTime = str(i["myExitTime"])

                # se escriben los datos
                row = weekday + [yeardaySin, yeardayCos, minuteSin, minuteCos, lastEx
itTime, queueLength, booths, myExitTime]
                writer.writerow(row)

def main():
    """
```

```
    Este script convierte los datos desde el formato que tendrían al salir de RPis,
    en datos de entrada para algoritmos de machine learning.

    Formato
    -------
        día de la semana; seno del día del año; coseno del día del año;
        seno de la hora y minuto; coseno de la hora y minuto;
        tiempo de espera de la última persona en salir de la cola;
        número de personas en cola; número de mostradores abiertos;
        tiempo que estaría en cola una persona que entrase ahora

        día de la semana: one-hot vector con un valor para cada día
            [1 0 0 0 0 0 0] Lunes
            [0 1 0 0 0 0 0] Martes ...

            para codificarlo se hará como 7 features:
            1;0;0;0;0;0;0 Lunes
            0;1;0;0;0;0;0 Martes


        día del año: número de dia 1-366, se calcula seno y coseno que
            darán un valor resultante entre -1 y 1

        hora y minuto: se calcula el minuto del día (hora*60+minuto) y
            con el valor resultante se calcula seno y coseno que darán
            un valor entre -1 y 1

        tiempo de espera de la última persona en salir de la cola: en minutos

        número de personas en cola

        número de mostradores abiertos

        tiempo que estaría en cola una persona que entrase ahora: en minutos
    """
    queue = []
    format_input(queue)
    output = get_output(queue)
    write_output(output)

if __name__ == "__main__":
    main()
```

Código B.2 *extract_embeddings.py*

B.2. Código para la predicción del tiempo de espera en cola en un aeropuerto

Este script de python utiliza una red neuronal para intentar predecir el tiempo de espera en una cola de un aeropuerto, a partir de los datos de entrada obtenidos anteriormente. En este script se basan las pruebas realizadas en los apartados: 5.4.2 y 5.5; en concreto es el utilizado en las condiciones mencionadas en el apartado 5.5.3.

```
import time
import pickle
import numpy as np
import pandas as pd
```

```
from sklearn.model_selection import KFold
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler

# Leer datos del fichero .csv que contiene los datos generados previamente
queue = pd.read_csv('LAX-international-YEAR-toML.csv', delimiter=";"
                    , names =   ["day(1)",
                                 "day(2)",
                                 "day(3)",
                                 "day(4)",
                                 "day(5)",
                                 "day(6)",
                                 "day(7)",
                                 "yeardaySin",
                                 "yeardayCos",
                                 "minuteSin",
                                 "minuteCos",
                                 "lastExitTime",
                                 "queueLength",
                                 "booths",
                                 "myExitTime"])

# Almacenar los datos en variables para poder utilizarlos luego.
# La variable X corresponde a los datos de entrada a la red neuronal, que
# consiste en todo el conjunto de datos, excepto el tiempo de espera en la cola.
X = queue.drop('myExitTime', axis=1)
# La variable y corresponde a los datos de salida de la red neuronal (el valor
# que se quiere obtener), en este caso es el tiempo de espera en cola.
y = queue['myExitTime'].to_numpy()

# Se normalizan los datos de entrada, para que tengan media cero y varianza
# unitaria, esto permite que la red neuronal converja más rápido.
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Se inicializa el objeto cv, que permitirá separar los datos en 10 partes
# iguales, de esta manera se podrá realizar una validación cruzada con 10
# conjuntos de datos (9 grupos a la vez para entrenar y uno para validar;
# repitiendo 10 veces el proceso).
cv = KFold(n_splits=10, shuffle=True)

# La list mean_relative_error_list almacenará todos los errores relativos
# medios obtenidos en las 10 pruebas de la validación cruzada.
mean_relative_error_list = []

# La list run_time almacenará el tiempo de ejecución de cada una de las 10
# pruebas de validación cruzada.
run_time = []

# Se realizará un bucle que iterará sobre las 10 pruebas de validación cruzada,
# cambiando el grupo que se utiliza como grupo de validación en cada iteración).
for train_index, test_index in cv.split(X):
    # Se guarda el tiempo actual para calcular posteriormente el tiempo que ha
    # durado esta prueba.
    start_time = time.time()

    # Se separan los datos, tanto de entrada como de salida, en los grupos que
    # se utilizarán para entrenar y validar.
    X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[train_inde
x], y[test_index]

    # Se guarda el modelo que se quiere utilizar en la variable mlp, en este
    # caso una red neuronal con múltiples capas, enfocada a la regresión.
    mlp = MLPRegressor(hidden_layer_sizes=(55,55), learning_rate_init=0.001, alpha
=0, max_iter=1000)
```

```
    # Se entrena la red neuronal con los datos seleccionados para ello.
    mlp.fit(X_train,y_train)

    # Se utiliza la red neuronal entrenada para predecir los valores de
    # salida, a partir de los valores de entrada seleccionados para la
    # validación.
    predictions = mlp.predict(X_test)

    # Se calcula el error relativo medio de todos los valores de validación.
    # Esto se hace con los valores de salida guardados previamente para
    # validación y los valores obtenidos con la red neuronal en el paso
    # anterior.
    mean_relative_error = (np.absolute(y_test-
predictions)/y_test).sum()/float(y_test.size)

    # Se almacena e imprime por pantalla los resultados, tanto el error
    # relativo medio, como los tiempos de ejecución.
    mean_relative_error_list.append(mean_relative_error)
    run_time.append(time.time() - start_time)

    print('Current results: ' + str(mean_relative_error_list))
    print('Time: ' + str(run_time))

# Se calcula el error relativo medio de las diez pruebas realizadas.
mean_results = np.mean(mean_relative_error_list)

# Se guardan todos los resultados obtenidos en un fichero pickle.
output = {'curr_results': mean_relative_error_list, 'time': run_time, 'mean': mean
_results}
f = open('NN_results_2x55_lr0.001_alpha0.pckl', 'wb')
pickle.dump(output, f)
f.close()
```

Código B.3 *NN_sklearn.py*

ANEXO C. CONTADOR DE PERSONAS EN COLA

En este apartado se mostrará el funcionamiento de un posible modelo, que podría ser desarrollado para calcular el número de personas en cola, así como los otros parámetros necesarios para la red neuronal que predice el tiempo de espera en cola.

Este modelo en su estado actual, es una propuesta básica que cuenta el número de personas que entran en una cola (no las que salen) utilizando sensores de ultrasonidos para detectar el paso de una persona y también una cámara para asegurarse que lo que entra a la cola sea una persona (no, por ejemplo, un carro de maletas).

Para la implementación de este modelo se utiliza una Raspberry Pi, juntamente con un script de Python.

C.1. Código para el conteo de personas

El script de Python es el siguiente:

```
# importar los paquetes necesarios
from imutils.video import VideoStream
import RPi.GPIO as GPIO
import numpy as np
import imutils
import time
import cv2
import os

print("[INFO] iniciando sensor de ultrasonidos...")
# pin que utilizará el trigger del sensor de ultrasonidos
TRIG = 23

# pin que utilizará el echo del sensor de ultrasonidos
ECHO = 24

# se escoge el modo de selección de pines BCM
GPIO.setmode(GPIO.BCM)

# el pin del trigger será de salida
GPIO.setup(TRIG, GPIO.OUT)

# el pin de echo será de entrada
GPIO.setup(ECHO, GPIO.IN)

# cargar el detector facial
print("[INFO] cargando detector facial...")
protoPath = os.path.sep.join(["face_detection_model", "deploy.prototxt"])
modelPath = os.path.sep.join(["face_detection_model",
    "res10_300x300_ssd_iter_140000.caffemodel"])
detector = cv2.dnn.readNetFromCaffe(protoPath, modelPath)

# inicializar el vídeo, luego dejar que el sensor de la cámara se caliente
print("[INFO] iniciando video...")
```

```
vs = VideoStream(src=0).start()
time.sleep(2.0)


# función que utiliza el sensor de ultrasonidos para medir la distancia
# devuelve la distancia medida en centímetros
def medirDistancia():
    # apagar el pin del trigger
    GPIO.output(TRIG, GPIO.LOW)
    time.sleep(0.1)

    # activar el pin del trigger durante 10 microsegundos
    # después volverlo a apagar. De esta manera el trigger envía el pulso
    # que el echo debe detectar
    GPIO.output(TRIG, GPIO.HIGH)
    time.sleep(0.00001)
    GPIO.output(TRIG, GPIO.LOW)

    # una vez enviado el pulso se espera a que el echo lo detecte
    # el último instante de tiempo que se detecta cuando el echo todavía
    # no ha detectado el pulso es el que se utilizará como tiempo inicial
    while GPIO.input(ECHO) == GPIO.LOW:
        pulso_inicio = time.time()

    # una vez el echo detecta el pulso se va midiendo el tiempo hasta
    # que deje de detectarlo, entonces se utiliza el último tiempo medido
    # para calcular la duración del pulso
    while GPIO.input(ECHO) == GPIO.HIGH:
        pulso_fin = time.time()

    # se utiliza el inicio y el final del pulso para medir su duración
    duracion = pulso_fin - pulso_inicio

    # utilizando la duración del pulso del echo y la velocidad del sonido
    # se puede calcular la distancia recorrida.
    # se debe tener en cuenta que el pulso ultrasónico debe ir y volver del
    # obstáculo, por eso se divide entre dos la distancia
    distancia = (34300 * duracion) / 2

    # se devuelve redondeada a centímetros, ya que no se necesita una gran
    # precisión
    return round(distancia)

# función que detecta si en la imagen que capta la cámara en
# este instante, hay una cara de persona (devuelve True) o no
# (devuelve False)
def detectarCara():
    # seleccionar un fotograma del vídeo
    frame = vs.read()

    # redimensionar el fotograma para que tenga un ancho
    # de 600 píxeles (manteniendo la relación dimensional), y
    # luego guardar las dimensiones de la imagen
    frame = imutils.resize(frame, width=600)
    (h, w) = frame.shape[:2]

    # construir un BLOB (objeto binario grande) a partir
    # de la imagen
    imageBlob = cv2.dnn.blobFromImage(
        cv2.resize(frame, (300, 300)), 1.0, (300, 300),
        (104.0, 177.0, 123.0), swapRB=False, crop=False)

    # aplicar el detector facial de deep learning de OpenCV para
    # localizar caras el fotograma de entrada
    detector.setInput(imageBlob)
    detections = detector.forward()
```

```
    # iterar sobre las caras detectadas
    for i in range(0, detections.shape[2]):
        # extraer la probabilidad asociada a las detecciones
        confidence = detections[0, 0, i, 2]

        # si la probabilidad asociacada a la detección
        # es elevada, se devuelve True
        if confidence > 0.5:
            return True

    # si no se encuentra ninguna detección con una probabilidad
    # asociada elevada, se devuelve False
    return False

try:
    # contador para saber cuanta gente ha entrado a la cola
    contadorEntrada = 0

    while True:
        # se mide la distancia, si es inferior a 70 cm, se considera
        # que ha pasado una persona
        if medirDistancia() < 70:
            # se realiza la detección facial, para saber si lo que ha pasado
            # es una persona o no
            if detectarCara():
                # si lo que ha pasado es una persona, se incrementa el contador
                # de gente en la cola
                contadorEntrada += 1
                print("[INFO] numero de personas en cola: " + str(contadorEntrada))

# para poder salir del bucle y cerrar correctamente el programa se deben pulsar
# las teclas Ctrl + C del teclado
except KeyboardInterrupt:
    # cerrando y limpiando, tanto el sensor como la cámara
    GPIO.cleanup()
    cv2.destroyAllWindows()
    vs.stop()
```

Código C.1 *contador_entrada.py*

C.2. Esquema del montaje del circuito

El esquema del circuito necesario para implementar este modelo es el siguiente:

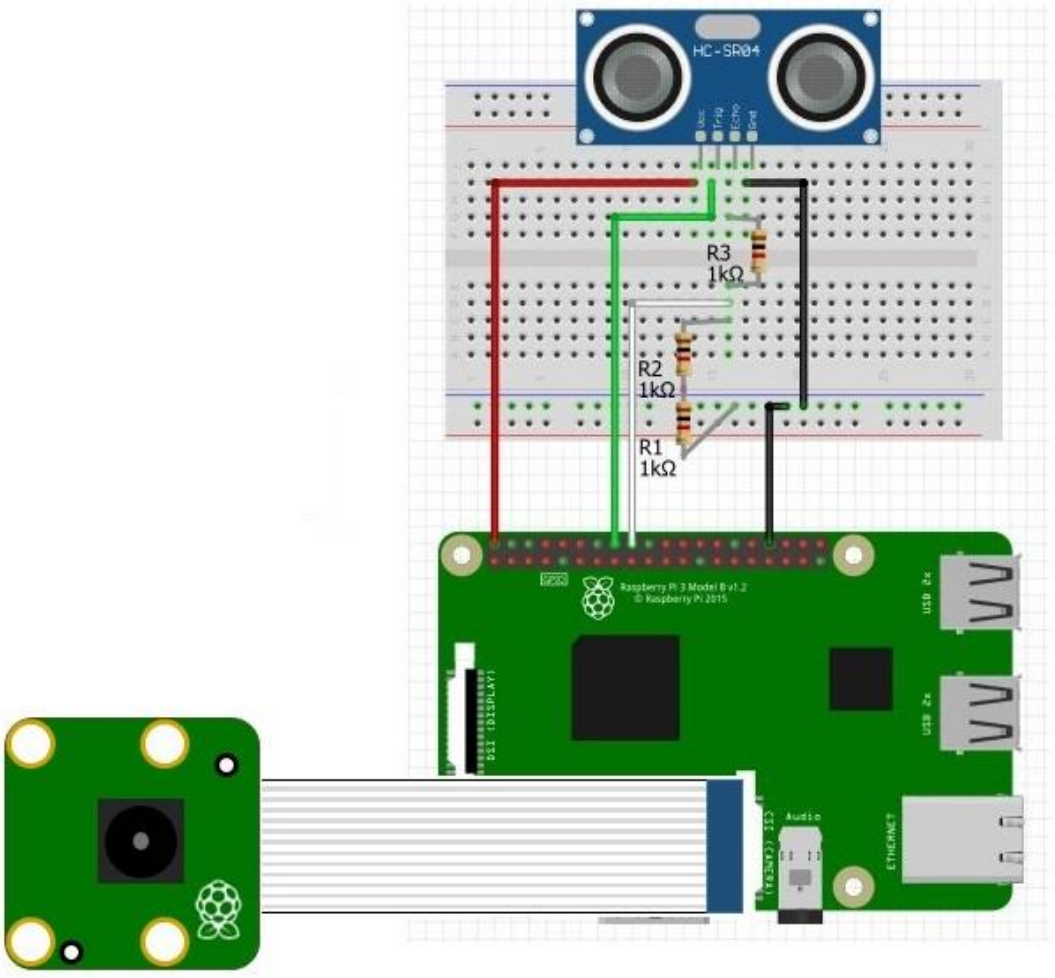

Fig. C.1 Esquema del circuito para el contador de personas